
From: "Dawson" <engler@Stanford.EDU>
Newsgroups: su.class.cs240
Sent: Friday, October 21, 2005 3:46 PM
Subject: lecture notes

here's a partial set of notes. missing the boehm and rinard papers.

****
****
****
****
*
*
* Worse is better notes
*
*
*

1. How does Gabrial rank correctnes, consistency, completenes, and correctness for the two approaches?

the 100% solution versus the 90% solution (where does elegance fit in?) (which would you prefer as consumer? as producer? why might production favor producer?)

the right thing:
correctness & consistency
|
completeness
|
simple interface
|
simple implementation

inferred:
- must be correct
- must be consistent, can be less simple and less complete to avoid inconsistency)
- simplicity not allowed to overely reduce completeness
- more important for interface to be simple versus implementatation

worse is better:
worse is better: almost exclusively concerned with implementation simplicity, and will work on correctness, consistency and completeness only enough to get the job done.

implementation simplicity
|
interface simplicity
|
correct
|
consistency
|
completeness
|
consisteny of interface

slightly better to be sipmle than corect
must not be overly inconsistent: can drop parts that introduce either complexity or inconsistency
completeness can be sacriieced for any other quality (however, contradict? consistency can be sacrificed to achieve completeness -- hacks)

good enough.

key feature: many cases something is better than nothing. so having a sort of right solution is much much better than the years of having nothing waiting for the right thing.

another: networking effect. if there is a non-trivial adoption cost, then the more people that use it the more incentive for other people to use it. language, OSes, chips. so you want to be the defacto standard if possible and then improve.

----

msoft model?
- backwards compatibility (not simple, not correct, not consistent)
- features (completeness)

why is complexity good?

why might releasing a few not-perfect systems before the perfect one be better? quick, charge each time, adapt to feedback.

----

handle all corner cases vs handle the common case.
reduces which costs?
- risk (simple thing out quick, modifications in direct response to feedback)
- time (out quick)
- cost (simple thing)

counter example sort of: the alpha chip. very much a 90% solution. but also a right thing approach to simplicity; because it was so simple could make blazingly fast.

what is the most important thing for worse-is-better?

2. Why is pc-lusering such a bad example?

1. it isn't right.
2. even if it is right, what is the trivial fix?

3. why sacrifice correctness? (follow up: what does correctness cost with current design practices? (time))

why sacrifice implementation simplicity? (why is it in intels and msofts interest to design incredibly complex interfaces and implementations?)

is there any piece of code you've used that is correct?
(almost certainly not)

----

1. As Gabriel states it, what are the features of worse is better vs the right thing?

the right thing: simplicity & correctuess & consistency & completeness are more or less equal.

worse is better: almost exclusively concerned with implementation simplicity, and will work on correctness, consistency and completeness only enough to get the job done.

2. What are some famous principles that contain/are contained in "worse is better?"

Do-it-all-at-once (right thing) vs piecemeal growth from something simple that grows in response to needs. Lots of examples of large systems in the latter:
- Internet
- web
- phone network
- gov't
- evolution (some of you are the right thing, most are new jersey)

[anything big that grew all or nothing?]

nice feature: feedback rather than having to understand a priori.

Successful, complex system: will be around for a long time.

Problem 1: have to live with mistakes.
Problem 2: correct choices will become mistakes.

Technology tradeoffs change dramatically, putting lots of pressure to modify - unfortunately, if successful, change is difficult because of installed base.

kiss: "do it right vs do it simple". One example: dynamic allocation vs fixed sized. The right thing would be tailoring everything to the right size. This has lots of costs, so in many cases you use a fixed size (or set of fixed sizes) (machine instructions, money, [s,m,l] vs tailoring])

"get something out the door quickly"

Note: right thing vs worse is better is a continuum rather than black/white. For example, if it crashes twice a week, but halves the code size, should you do it?

He says this is a charicature:
What are the schemes that these distort?

1. worse: concerned with finite resources and getting useful out the door. Incompleteness to get a working system.

2. better: assumes sufficient resources; want a complete system. wants good algorithm from the literature. Completeness.

Pragmatics vs perfectionist?

----

1. give example of worse things and better things?

do these really fit with Gabriel's thing? what other things are going on? what kind of different slants can we give it?

happens with spoken languages --- English defacto standard, other more elegant languages non-starters.

2. why is worse is better better?

acceptance to the widest possible market and then improve. non-trivial adoption cost, then go (e.g., don't really see this with music (you buy vanilla ice and then stop), but do with things like languages, etc)

lets say the right thing is X, and you can get there either by doing the right thing or by doing worse is better. which way would you expect to make potentially (significantly) more money?

3. is there a better characterization than worse is better? Microsoft seems to show that you can take worse even further. correctness and simplicity not necessary at all. in fact many successful things show that simplicity is not really all that important: Perl, c++, x86, ...

4. advantages of incremental versus all or nothing?

- old stuff still works. MIPS vs x86 (simple versus complex and horrible, but non-trivial switching cost)

- iteration: get something out fast, find out what is wrong with it, and do something. feedback is key. often, don't know how to solve the problem until you've solved it. here you can get something out the door, get people using it (hard to switch) then adapt to what they want.

5. What important variables does Gabriel ignore?

Cost is one big one, both in terms of how much you charge to sell, and how much costs to build.

For example, lets say you have $50 bucks to make a new operating system, how should we weigh simplicity, correctness, etc?

time is another.

risk.

Will a free system have better survival/promulgation features than proprietary?

6. Is Gabriel right? is worse better?

after ten years, he doesn't actually know: wrote a series of papers:

worse is better (pro)
"worse is better is worse" (con)
"is worse really better?" (pro)
models of software acceptance (pro)
"is worse (still) better?" (con) (position paper)
"worse (still) is better!" (month later, pro)
still can't decide.

---

is a continuum (a relative term) rather than absolute
worse better
linux bsd

windows linux (well, sort of)
unix multics --- but unix is much more elegant.

argument is dated: we have windows (worst is best) vs unix (arguably the right thing)

in my mind, the most useful thing about this is to make pragmatic, expedient choices: cut corners initially, get something out, do the parts that matter righ.

---

the intersection effect: large set of poeple, have to appeal, number of things is quite small.

music, tv, books.

japanese cars: initially cheap, badly made, wimpy. but took over from more fancy "right thing" ones (compared to BMW, porche, say)

implementation simplicity not the worse-is-better --- use it to give yourself a lead time; works well for industry standard setters.

c++, modula3, eiffel, smalltalk, CLOS

[java seems reasonable]

---

key things:

1. this class is mainly about discussion. if i have to listen to some guy talk for 50 minutes, i start zoning out after 10 and then start skipping classes. can't really learn either, just passive. going to try to make it much more interactive, which is a lot of work for us.

2. going to be an experiment to turn a large class into discussion, so we're going to have to be experimenting with what works and what does not. the class will roughtly be 60% from tests: you take 3 and pick the highest two. 40% will be everything else, which will include class participation, pop quizes and possibly pop presentations.

sitn students don't have to worry, it's harder to participate so we'll just have you write

so: everyday what should you do? read the paper for the next class at least twice closely. make notes on its structure, what the main points are, try to think of examples, counter examples, whether you believe it, what is cool, what is broken.

(In general, things that really succeed seem to be ugly. perl,

tcl, c++, msoft products. vs python/ruby/scheme.)

networking affect: non-trivial fixed cost to getting in, want to be able to talk to other people.

* So can we come up with a better example of Worse is Better?
(1. The Y2K leap year problem may be suitable. To recap, you are supposed to have a leap year when the year number is divisible by 4, but skip those years that are divisible by 100, but don't skip the leap year if it is divisible by 400. Thus 2000 *is* a leap year.
The "worse is better" school says that we just calculate the year mod 4, and if the result is zero we have a leap year. This calculation gives the right answer from the dawn of computing through 2099 and after four years of field experience the implementation is unlikely to have any bugs in it.
The "do it right" school says that we do a bunch of extra arithmetic to check for divisible by 100 and divisible by 400, with various conditionals to decide which case we are presented with. Some of this code will be exercised for the first time ever at the end of this month, at which time some, but not necessarily all, of the remaining coding bugs will emerge. So "worse is better" seems to have an identifiable advantage, at least for a century or so.)

The argument is an evolutionary one: viruses that spread quickly and are pervasive. if successful they will be improved.

good drives out excellent, the most popular is the least good

---

[according to gabriel]

characteristics:
1. impl should be fast.
2. should be small.
3. should interoperate with the programs people use
4. should be bug free, and if that means fewer features so be it.
5. use few abstractions. (abstraction = page faults)

implication: far better to have an underfeatured product that is rock solid, fast and small than one that covers what an expert would consider complete requirements. [so what is microsofts model?]

benefits
- less development time: out early, adopted as defacto standard
- can run on smallest computers; probably easy to port.
- pressure to improve over time will acquire the right feature -- thos the customers/users want rather than those the develops

thik they should have.

path of acceptance: act like a virus, providing functionality with minimal acceptance cost. can be improved and is small and simple enough to do so (nice point: small simiple = much much easier to modify in response to feedback.)

small simple = quick to build, cheap to build, easy to incorporate feedback, customization (feature of early unix)

the acceptance model for the right thing is that it comes late to market but is so wonderful it is accepted. has to run on every platform right away or quickly. can happen; unlikely.

right thing based on the philosphy of letting the experts do their expert thing all the way til the end until the users get their hands on it [key point about feedback]

incremental releasess: can charge for each, will be a smaller steop, with less investment, less time, more feedback: more of the "right thing" from the users point of view (perhaps — can always correct if not)

key idea: identify the true value of what you bring to market, test ideas in small context, imporve in participatory way, and them package in the elast risky way

---

most popular thing = mediocre. might not be anymore more deep.

in some sense, successful will get ugly: will have many demands, will likely adapt to them (success), get more random entrpy.

why earliest adopted so good? 1Billion people speak english, 10 speak esperanto. which would you probably learn?

key: something is better than nothing.

---

when counter examples?
qsort: simple, clean, elegant. much of mathematics is like that. elegant at least, if complex. engineering has costs.

accessible. this is the main thing? if it's out there its accessible. if it's ported to many places. if ubiquitous.

the more accessbile, the more potential consumer base, and the stronger the final networking effect.

****

```
**********************************************************************
**********************************************************************
**********************************************************************
*
*
*
* Therac notes.
*
*
*
```

Who are these authors? Bias?
"software eng, safety engineering, and gov't and user standards"
as a solution to the problem.

[From Saltzer]

So how do these Therac things work?

```
                                     photon
turntable position:   electron therapy   (x-ray)   field-light
                      mode               mode      mode
beam energy:          5-25 MEV           25 MEV    0
beam current:         low                high      0
beam modifiers:       magnets            flattener none
```

If you somehow get the beam current set to high at the same time that the turntable is positioned with magnets rather than with the flattener, you have set the stage for a disaster, with a high X-ray beam current delivered directly to the patient, rather than being attenuated by a factor of 1000 or more. Another way to kill a patient is to turn the beam on with the turntable in field-light mode.

An interlock is needed. The hardware interlocks of the Therac-20 were replaced with software interlocks. What is a software interlock, anyway?

(The "software interlock" is just a boolean flag for which a non-zero value indicates that there is reason to believe that the turntable position should be verified for consistency with the beam setting.)

What are the actual details of the two race conditions?

Heisenbugs vs Bohr bugs:
Bohr:
\+ always happen so easy to track down
\- always happen so cannot use duplication or reset to

get around.

Heisen:
\- hard to track down
\+ can always retry, often won't happen.

Things that led to bad things, hid them, made them more difficult to track down, encouraged, or caused them:

- Not entirely companies fault: there were environmental factors incentives to suppress information. For example, company X and Y have the same error rate, but Y reports it all, and X does not. FDA more likely to go after Y, customers more likely to think it sucks, more lawsuits even when its not at fault.

- written in PDP 11 assembly language! (p20)

- lack of documentation or spec (p20)

- concurent access to shared memory without real synchronization. led to lots of cliched test-set race conditions (p21):

```
if(!x)
x = 7;
```

- lack of mechanism in AECL to follow up reports of suspected accidents.

- patients get 10-100x more radiation than asked for; underdosing also a problem, which means that people are essentially untreated.

- during knnestone incident, printout feature was disabled, so no hard copy of treatment data (p23)

- regulations do not require health care inst to report accidents (only manufactures). As a result, less than 1% reported (p23). Problem fixed in 1990.

- Race conditions so could not replicate (p26) so could not really track down.

- users did not learn promptly about accidents; usually had to figure out from other users. (p23)

- opaque error messages ("h-tilt", "malfunction 54") without any indication that an error was really dangerous. manual does not explain or address them. or wrong: "dose input 2" = too high or too low; then used to treat patients for the rest of the day!).

- no defensive checks to see if parameters entered are out of

range (p24)

- wrong display: "no dose": when it actually had. (p23)

- accustomed to frequent malfunctions that did not seem to have any serious side-effects. (p23) "insensitive" to machine malfunctions (p24) commonplace. the texas operator was used to machine stopping or delaying treatment.

- texas: video display unplugged, audio broken, so couldn't hear patient (screaming). got second overdose.

- users told: "so many safety mechanisms that virtually impossible to overdose" (p 24)

- dual mode machine so could mix modes (see table).

- substituted software checks rather than hardware interlocks.

- believed/said things like "improved by 5 orders of magnitude" which lets us infer they are idiots or evil.

- magic constants (suspend 5 times before you had to do system reset). reduced to 3 on (26)

- refusal to install potentiometer (26)

- lack of information propogation. interlocks labeled as redundant.

- lying/mistaken: claimed no other incidents when there had been. (27) texas said no other incidents ut must have known of the hamilton incident.

- software reuse, but misguided, since interlocks removed. look back and therac-20 had the same problems but was catching them (21) replicated errors; therac 20 was interlocking (can see when students use them with weird testing).

- general: letter did not say why it was so crucial to disable up arrow key, or that it was really urgent (31) [Good!: FDA said fix this!]

- bad sign: continued push back against testing requests by FDA (32)

- memory limits would not permit audit trail (37) and that users could not get source code.

- industry has been complacent because everything went well so far (38)

- did not report when something bad happened (which should have been easy).

- slow system: took almost three years from incident to stopage.

Good things that happened that led to checking:

- user group meeting.

- conservative approach in fault tree analysis, and interlocks (36)

- error had a physical manifestation: a reality check against virtual claims.

- turning off beam would not cause that much trouble; interlocks can be used.

- after july 30th (est 13K-17K), fda was informed.

- yakima incident: good follow up by hospital physist. e.g., reaction to kemo ruled out because no stripes. heating blanket ruled out because wire pattern (x-rayed!) did not correspond (p26)

- tyler texas physicist investigated aggressively.

- end to end checks

```
**********************************************************************
*
* Eraser Notes
*
*
```

How hard are these bugs to find and eliminate? Potential problems:

a) Timing dependences may make the bug difficult to reproduce. What is worse, the instrumentation people insert to help them find bugs may change the timing in such a way that the bug never shows up.
b) The bug is usually caused by the unexpected interaction of two loosely related pieces of code that are often in different modules. So the person debugging must understand the module interactions and cannot reason about the system one module at a time.
c) The manifestation of the bug may occur long after the execution of the code containing the bug.
d) The code that fails may be very far away from the code containing the bug.

---

what is the definition of a race condition?
- what's a source of false positives/false negatives in theirs?
- what's a better definition?

NOTE: if I acquire all locks before every load or store, and release them after, will get no error, but protect against no races.

their mental model:
every memory location has the set of locks used

what is the granularity of shared state?
- word can have a lock (don't protect bytes or bits: can produce false positives)

  - each word has a lockset index associated with it.

what does atom have to do?
- instrument lock/unlock
- add/remove lock from current lockset.
- has to know if read/write lock.
- has to know which parameter is the lock

- allocation: initialize shadow memory (need to do data segment at startup)

- insert a call to eraser on every load and store.

---

calls malloc: what happens:
allocates shadow memory as big as the allocation.
puts it in the virgin state
sets the thread id to the current thread (calls thread package)

atom puts in a call to this routine on every load store that is not off the stack pointer:

```
  void compute_transition(lockset *ls, void *addr, int op) {
# alpha has an 8K direct mapped cache --- what is a really bad
# value for offset?
i = ((unsigned)addr >> 2) + offset;

# virgin has no previous accesses.
 if s[i].state == virgin
s[i].state = exclusive
s[i].ls = thread_id.
# only rd/wr from cur thread
 else if s[i].state == exclusive
if(s[i].ls == thread_id)
# do nothing
else
if(write)
s[i].state = shared-modified;
else
s[i].state = shared;
s[i].ls = cur_ls;
```

```
 else if s[i].state == shared
s[i].ls = s[i].ls intersect cur_ls;
if(read)
# no error if goes to empty.
else
s[i].state = shared-modified;

 else if s[i].state == shared-modified
if(read)
s[i].ls = s[i].ls intersect all_locks_held;
else
s[i].ls = s[i].ls intersect all_write_locks_held;

    if(s[i].state == shared-modified && s[i].ls = {})
error "BOGUS";
  }
```

modifications:
1. if removed lock not there, complain.
2. if added lock already there, complain
3. if we are going to go to empty, emit warning, but leave in old lock tate.

what things do they gloss over?
+ atom already blew it up by 2x code size i believe.
+ granularity of protection 4bytes --- if you could protect 1 byte, then 4x more.

add in annotation support?
- eraserignoreon/off:
do not report --- this means they should not refine as well, otherwise it's not that useful.
- eraserreuse
reinitialize
- eraserreadlock/unlock/writelock/writeunlock: have to say what parameter is the lock (pass in address).

---

13) What does the experimental evaluation say about application characteristics and the utility of the tool?
  a) Altavista basically had no serious synchronization errors. There were false positives, but a small number of annotations removed them all.
  b) One bug fix in the Vesta cache server. The problem is related to the interaction of a standard synchronization idiom for machines with a sequentially consistent memory model and the weak memory consistency model in the Alpha. My guess is that when the code was written, it was not intended to run on machines with weak memory consistency models, then was ported to the Alpha without a reexamination. A common source of errors - see the Ariane rocket failure.

  c) Petal - no serious synchronization errors.
  d) Student programs - 10% of apparently working student programs had synchronization errors.

lines locks locksets annots errs
altavista
mhttpd: 5000lines 100 250 10 0
Ni2 20000 900 3600 9 0

vesta (cvs) 30K C++ 26 70 10 1
petal 25K C 2
[statistics: minor]

ugrads 10%

These programs are surprisingly free of synchronization errors. The data suggest that Eraser might not be useful in making production programs more reliable. Eraser might therefore be more appropriate as a tool that would make it easier and faster to find synchronization errors during program development. It would be interesting to see a bug fix log for these server programs to see if they had significant problems with synchronization errors during program development.

An alternate perspective is that developing thread-based programs may not be that difficult for very good programmers like the ones who developed these servers, or that the servers themselves do not use synchronization in a very complicated way, so it is straightforward to get it right.

[false: memory reuse, private locks, benign races]

Different from the text:

*On page 398, it says that "A write access from a new thread changes the state from Exclusive or Shared to the Shared-Modified state..." But figure 4 says that a write by any thread in the Shared state takes it to the Shared-Modified state. This is a contradiction. Which is right?

(Oops, a bug in the description. The figure is right. Looking at the later description of the implementation, any write will take it to shared-modified. Once it is shared it is running the lockset algorithm without giving warnings, which means that the per-variable shadow area contains the lockset pointer, so it can no longer be keeping track of the thread number of the original writer. We can also reason from what it should do. If anyone is writing into a variable that at least one other thread has been reading from, we have a possibility of a race, so we had better we raising alerts if the locking protocol is violated. [a legalistic reading of the text can claim that it is technically accurate; it is true that a write access from a new thread in the Shared state does take it to the Shared-Modified state; they just didn't bother to mention that a write access from the old thread in the Shared state also takes the variable to the Shared-Modified state. Under that interpretation the sin is that the authors forgot to mention one important case.])

* have them list all the false positives and false negatives that eraser gets.

---

why sem not a race? forces sequential execution:

```
x++;
v(sem);
p(sem);
x++
...
```

is the lockset a per-thread data structure? does it need to be?

---

start with:
How hard are these bugs to find and eliminate? Potential problems:
  a) Timing dependences may make the bug difficult to reproduce. What is worse, the instrumentation people insert to help them find bugs may change the timing in such a way that the bug never shows up.

[used to hate runing on a faster machine. different speeds; also different mem consistency models.]

insert a printf, it disappears.

  b) The bug is usually caused by the unexpected interaction of two loosely related pieces of code that are often in different modules. So the person debugging must understand the module interactions and cannot reason about the system one module at a time.

[violate modularity: have to look at all critical sections

```
lock(l);
x++;
unlock(l);
...
```

x is behaving strangely: can i just look here?
no i have to expand the ellipses.
]

  c) The manifestation of the bug may occur long after the execution of the code containing the bug.

d) The code that fails may be very far away from the code containing the bug.

What was the scope of the tool?
a) Threads that synchronize using only mutual exclusion locks (no condition variables).
b) Bugs that can be detected based on dynamic execution. So if there is a bug in a part of the program that is not executed, bug will not show up in that run.
c) Shared variables are either heap or global variables accessed by multiple threads.
d) If the programmer puts in synchronization, the granularity is assumed to be correct.

4) Basic assumption: the programmer has mentally associated each piece of data with a lock, and a correct program will hold that lock during every access to that piece of data.

5) What is the basic problem the Lockset algorithm addresses?
Determining the association of locks and data.

How does it solve this problem?
It dynamically constructs the set of locks that can be associated with each accessed memory location. This is computed as the intersection over all accesses to that memory location of the locks that the program holds when it performs the access.

How does a synchronization error show up?
If a lock set ever becomes empty, a synchronization error is reported. Note that the error itself does NOT have to occur in the program execution - just the possibility of an error.

---

13) What does the experimental evaluation say about application characteristics and the utility of the tool?
a) Altavista basically had no serious synchronization errors. There were false positives, but a small number of annotations removed them all.
b) One bug fix in the Vesta cache server. The problem is related to the interaction of a standard synchronization idiom for machines with a sequentially consistent memory model and the weak memory consistency model in the Alpha. My guess is that when the code was written, it was not intended to run on machines with weak memory consistency models, then was ported to the Alpha without a reexamination. A common source of errors - see the Ariane rocket failure.
c) Petal - no serious synchronization errors.

d) Student programs - 10% of apparently working student programs had synchronization errors.

15) It is interesting to compare Eraser to another tool with similar characteristics, Purify. Purify is designed to catch memory errors (dangling references, memory leaks) in C programs.
It has a lot of similarities to Eraser:
a) Uses binary rewriting.
b) Uses a dynamic approach to catching errors, which means it misses errors that are not exposed in the instrumented execution.
c) Designed to catch a very nasty class of bugs that cause programs to fail in mysterious ways.
d) A safe programming language like ML would eliminate the errors that Purify was designed to catch. Analogy with monitors: ML and other safe languages did not catch on, probably in part because the safety was too constraining. It prevented programmers from doing useful things like writing generating a write to a specific memory address or writing a general memory allocator. Interesting development: emergence of Java, which is a safe language.
Purify was a commercially successful product, which illustrates the importance of memory bugs in C programs.

16) Eraser illustrates several recurring areas of tension in programming tools:
a) Static versus dynamic error checking
b) Checking an unsafe language (with potential false negatives) as opposed to using a language whose model of computation eliminates the potential for errors to occur.
c) Doing analysis/instrumentation at the assembly level (this is getting increasingly popular) as opposed to the source language level.

KEY:

*Why the elaborate state diagram of figure 4 (page 398). Why not just use the first version of the lockset algorithm described on page 396?

(Because not every variable is both shared and modified, and it is only shared-modified variables that can be the source of races. So the state diagram shows a way discovering which variables are actually shared-modified.)

*On page 398, it says that "A write access from a new thread changes the state from Exclusive or Shared to the Shared-Modified state..." But figure 4 says that a write by any thread in the Shared state takes it to the Shared-Modified state. This is a contradiction. Which is right?

(Oops, a bug in the description. The figure is right. Looking at the later description of the implementation, any write will take it to shared-modified. Once it is shared it is running the lockset algorithm without giving warnings, which means that the per-variable shadow area contains the lockset pointer, so it can no longer be keeping track of the thread number of the original writer. We can also reason from what it should do. If anyone is writing into a variable that at least one other thread has been reading from, we have a possibility of a race, so we had better we raising alerts if the locking protocol is violated. [a legalistic reading of the text can claim that it is technically accurate; it is true that a write access from a new thread in the Shared state does take it to the Shared-Modified state; they just didn't bother to mention that a write access from the old thread in the Shared state also takes the variable to the Shared-Modified state. Under that interpretation the sin is that the authors forgot to mention one important case.])

*Show me an example in which we get a race if only one thread ever writes to the shared variable.

```
(

  thread 1            thread 2
                  x = 10 (initialization)

  acquire(xlock);
  if x > 5            x = 2
    y = x*3;
  else
    y = 0;
  release(xlock);

)
```

[don't get this]

*In section 3.4, it says that Eraser would have trouble with semaphores because they are not "owned". This takes us back to the earlier question: Does Eraser really depend on ownership?

(The state diagram of figure 4 has arcs labeled "first thread" and "new thread", so it certainly needs to know who is setting a lock. But presumably Erasericould find that out by looking in some currentÃ thread system variable. And it is certainly true that a locking protocol in which one thread acquires a lock and another thread releases it is going to be hard to debug. But it doesn't seem that Eraser would give different answers if the discipline of only the owner can release a lock is abandoned.)

---

10) How accurate is Eraser?
a) False negatives:
1) Dynamic initialization races that don't show up in the execution.
2) Errors in unexecuted pieces of code.
3) Dynamic lock addressing that may be correct in some runs but incorrect in others.

b) False Positives:
1) Phased computations that don't use synchronization for data that is read-only in a given phase.
2) Data that goes through an application-specific memory allocator and uses a different lock second time around.
3) Hierarchical locking strategies. Example:
Holding a lock on a tree node gives the program the right to modify any node in the subtree. Some programs may lock the tree at different granularities.
4) Alternative lock primitives that are not instrumented by Eraser.
5) Sometimes the data race is benign:
a) Computation requires only approximate, not exact information. So incrementing variables without synchronization is OK in some circumstances as long as errors don't show up too often.
b) Single reads and writes to words of memory are atomic. If the program only requires that level of atomicity, there is no need for locking.
Annotation mechanism to turn off false positives.
c) Not designed to handle:
1) Optimistic synchronization primitives.
2) Condition variables.

11) The utility of Eraser depends on
a) Frequency of false positives - too frequent makes the tool cumbersome to use.
b) Number of bugs that Eraser catches in practice, which depends on the number of bugs that programmers introduce into multithreaded applications and on how many of them Eraser catches.
c) Perceived severity of bugs that Eraser catches.
d) The number of applications that meet the Eraser model of synchronization.
All of these issues depend on application characteristics. So the experimental evaluation is absolutely crucial to understanding whether the tool is useful or not.

---

*On page 392 the authors say "Only the owner of a lock is allowed to release it." Is this true of the lock implemented in chapter 3 (page 3-62)?

(The implementation of chapter 3 certainly doesn't enforce any such restriction, though it would be easy to add it. Some locking systems enforce those semantics, others don't.)

*Does Eraser actually depend on this rule?

(It does need to know which thread is setting a lock, in order to run more advanced versions of the lock-set algorithm. But we haven't gotten that far yet, so let's bookmark that question.)

```
***********************************************************************
***********************************************************************
***********************************************************************
```

****
*
* Mesa Notes.
*
*
*
what can you say about paper?
- 20 years ago we wtill read.
- lots of citations ot other people. schlarly. including papers
we will read
- turing award winner.
- beautifully written.

experience paper: no real numbers, doesn't introduce ideas, lays out
design decisions. the "hard evidence" is the least useful part of
the paper (experience in real systems, the numbers are mostly curios)

How environment different?
in the same address space.
cooperative.
not time sliced.
microcoded instructions
weird stack.

----

using the following code:

```
    ------------------------------------------
int sem;
condition non_zero;

P()
if(!sem)
wait(non_zero);
sem--;

V()
sem++;
notify(non_zero);

will_block()
return sem != 0;
    ------------------------------------------
```

monitor:
- code + data + synch. written next to each other.

- data (sem) can only be modified by monitor itself.

difference between monitor and module: 2 bytes of data
and 2 bytes of code. what is the code? what is the data?
(data is lock structure: seems to have 1 bit for a lock,
and 15 bits for the tail. so there can be 2^15 bytes

of pointers).

- types of routines:
entry: can be called from outside. acq/rel lock.
procedure -> entry adds 8 bytes of code.
monitorentry, monitorexit, not sure what the
other is...

internal: called only from inside. no acq/rel.

external: does not acquire lock. Why? are these just
speed hacks or is there some correctness? cannot
wait, cannot call internal routines.

guesses as to why wait adds 12 bytes of code? probably
includes inline queue manipulation, including a check.

how many queues in this code?
- one for monitor lock, one for condition variable.

- all the ways to get switched out in this code?
- return
- wait()
- another thread becomes higher priority: scheduler
switches us out at any point.

- generate an exception that you handle: will not be
released if you return without handling exception!

what is a notify going to do mechanically? pull something
from the head, if anything. broadcast puts everyone on the
ready queue.

lock [can figure out from table]

```
struct {
lock;
queue *tail;
};
```

condition variable:
can be broadcast or notified.

```
struct {
queue *tail;
int timeout;
int wakeup-waiting-switch;
};
```

process
thread of control

can be joined, or detached.

contains: stack of frames, plus a 10 byte descriptor
(ProcessState: kept in a fixed located table; size determines
the maximum).

runs on a frame: frames seem to grow pretty weirdly. taken
from a heap, rather than pushed/popped. why is that?

* when are switched?
wait on lock on entry.
someone else becomes higher priority
wait on condition.
possibly when you do a notify, waking up a higher
priority thread.

no timesliceing means what? [entirely driven
by when process relinquishes, or makes another
guy have a higher priority.]

* woken up : how?
- notify (cond);
- timeout -- associated with each condition variable.
- abort[p]: pass process descriptor.
- broadcsat(cond): wake up everone.

- can do arbitrary wakeups at arbitrary points and
they should work correctly.

* wait:
- releases lock of containing monitor.
- does not not release locks above.

* notify:
* does not release control of monitor lock

- puts notified thread, if any on ready queue.

- hoare: switch immediately. know that whatever was being
signalled is done.

mesa: resume at some convenient time. nice feature: can
replace with broadcast, which allows easier reasoning.

"verification is actually made simpler and more
localized." p 11 why?

How does having mesa-style monitor semantics actuall
help things? [Correct can always be woken up, even

when not necessary.]

* naked notify: outside of monitor. what is the problem?
what is an alternative?

what happens if we signal a higher priority process?
- sucks: put on ready queue, will preempt us,
will block on lock, will wake us back up.

* scheduler: premptive between priorities, fifo within.

----

what are all the queues in the system?

one central ready queueu, and then one with each lock instance
(monitor lock, condition), and a fault queue. the first pointer
is to the *tail*. this lets you use one cell of storage to add
processes to end, as well as cycling through.

features of q?
- same priority, fast insertion, will be fifo
- high priority fast
- low priority fast
- take highest off queue fast

What is a monitor invariant?

- just the abstract data structure invariant, but protected
against concurrency.

examples:

- that the availablestorage = the actual bytes.
- that some freelist actually holds freed data.
...

----

how is mesa the Right Thing?
- check every instruction

- have run queue logic in *hardware* (microcode) not sure if
right thing or not.

how New jersey?

- frequency of use -> put in hw rather than the most elegant partition
- wait semantics: since it's a hint, what do they get out of it?
can do caorse conditoins.

- the fact that they disable interrupts!

- why not have recursive monitors? should preserve invariant.
- des not protect against dangling refs.
- the locks with clause: does not check that you do not modify via aliasing. how could you check?

- don't worry too much about fairness: in a properly designed system there shouldn't be too many processes waiting for a lock.
- fixed size processState array.

- non-recursive locks.

Tradeoffs between eraser and mesa?
- eraser can find when you forget to wrap shared state in monitor.

- monitor does sugar of acquisition and release cleanly.

- limits what you can express — have to do monitor.

- completely prevents classes of errors instead of kind of finding them.

- languade independent.

- prevents a larger class of race conditions: eraser ensures that you have same lock when you modify same variable. will not prevent you from doing something stupid. do these look the same from the point of view of eraser? do they have the same semantics?

```
lock(l);
x = x + 1;
unlock(l);

lock(l);
tmp = x + 1;
unlock(l);

lock(l);
x = tmp;
unlock(l);
```

- could make something an external that you really shouldn't.

How can you deadlock?

```
        ------------------------------------------
 entry foo(int x)
if(x) foo();
        ------------------------------------------

 entry foo()   entry bar()
bar(); foo();
```

```
        ------------------------------------------

 entry foo()   entry bar()
bar(); wait(c);

  ------------------------------------------------
 entry foo()   entry bar()
bar(); throw invalid;


  fork foo();
  fork foo();
```

what happens? will go to debugger. holds lock still held, second one will deadlock. why did they do this? modularity! otherwise return with monitor invariant all screwed up.

why don't they release the lock?
violates modularity: don't know if the person you are calling could call you.

---

questions:

what is a monitor invarient?

Your ex-140 partner loudly declares that if the semantics of the ``wakeup-waiting switch'' is good for naked notifies, it must be good for normal notifies, which should be replaced with them. Can you do this substitution and preserve correctness? (List any assumptions you must make.)

> The replacement should have no effect for well-formed mesa programs that use a while loop to recheck their wait condition. Such programs will work (albeit possibly more slowly) even in the presence of completely random wakeups.

Would a Mesa programmer have any use for a Mesa version of Eraser?
To detect deadlock, to detect when you should be using a monitor.

Give two examples where Mesa makes a ``New Jersey'' style decision. Wakeup semantics. No recursive monitors. ``Locks with clause'' object not protected from modification. Does not worry much about fairness on locks ``in a properly designed system should not be many processes waiting for locks.''

Explain from the Mesa paper: ``...[while] any procedure suitable for forking can be called sequentially, the converse is not true.''

Uncaught exceptions in forked procedure causes system to go to debugger.

```
******************************************************************************
*
*   Capriccio  Notes.
*
*
```

What's wrong with user-level threads?
 [picture: many uthreads and one vCPU]
 A blocking system call blocks all threads.
 Why not use select() to avoid blocking?
  disk read. open(). page fault.
 Hard to run as many threads as CPUs.

What's wrong with kernel threads?
 [picture: many uthreads, one vCPU for each]
 Handles blocking system calls well.
 10x-30x slower than user threads, due to kernel calls.
 Which operations have to call into the kernel?
  Thread creation?
  Thread context switch?
  Waiting for a held lock?
  Waiting for a free lock? (maybe not...)
  Releasing a lock?

What about multiplexing user threads on kernel threads?
 This is what the paper mostly compares against.
 Viewing kthreads not as the feature, but as hidden machinery.
 Try to do most operations in user level: create, ctx switch, locks.
 Kernel can't know which is the right uthread (ie kthread) to run.
 Kernel may pre-empt during a critical section.
  Assuming user-level locks.
 Kernel may not understand priorities of uthreads.
 Bad to have fewer runnable kthreads than CPUs:
  Wasting CPU time.
  This will happen when kthreads block in the kernel (page fault).
  So spawn a few extra kthreads?
 Bad to have more runnable kthreads than CPUs:
  Scheduling/priority and pre-empt in critical section.
  Also caused by multiple unrelated jobs competing for CPUs.
 Summary: kthread *not* a virtual CPU!

---

ousterhout threads argument:
mendel's advisor. was a prof at berkeley. did sprite, systems guy. then did tcl and quit.

this was an invited talk. you invite someone famous. they talk for about an hour. many talks aren't so good. the good ones

tend to be about experience. you might not know from text but this was probably the most widely cited (influential?) invited talk in systems.

main claim: most things, events better.

what is an event? (a closure)
pointer to code.
values for parameters.
values for other state.

foo(a,b,c) becomes something like:

```
struct {
void (*fp)(T,T,T);
T arg[MAXARGS];
} e;

e.fp = foo;
e.arg[0] = a;
e.arg[1] = b;
e.arg[2] = c;
```

threads =
* pre-emptive scheduling of different entities.
synchronize.
monitors = low concurrency
fine-grain = major complexity
deadlocks.

event =
* non-preemptive.
[is orthongonal though: can change one or the other and the bulk of the code isn't modified]

one execution stream.
register callbacks for when event happens
event loop waits for events, invokes handler (duality paper)
generally short lived.

GUI: window events
servers: one handler for each message type.
event driven i/o for overlap.

problems:
- long running handlers = non-responsive.
[cooperative problem: have to wait for yeild.]
- local state across events painful
[have to manually wrap up: or you could have scheme support.]

- no concurrency

[can of course combine, but then lose cooperative which capriccio techniques still work? need to do M:N.]

- event i/o not always well supported. (and you have to know when could happen)

[have to put in wrappers. need async. but you can use kernel threads! just have a helper issue the request]

pros:
+ debuggging only related to event order, not schecduling

order of threads
easier problems: slow vs corrupted mem.

+ minimize concurrency

+ faster on single CPU (no locking, no context switching)
+ more portable: just rip stack. no context switching code.

threads:
long running handlers (processes!)
true concurrency

what does he mean about high-end servers?
[just means multi-processor, and you want to use them]

what are thread good at?
automatic stack management
what are events good at?
cooperative.

easy to combine both.

---

usenix fibers. nice paper. might be too simple for class however, unless we throw in the hotos and ousterhout talk.

main point:
can seperate manual stack management from auto, pre-emptive from non-preemptive. sweet spot: non-preemptive threads.

task management: cooperative or pre-emptive. special case: serial task management which runs tasks to completion before doing the next one.

no conflict of shared state, no one can violate invariants. doesn't work so well when task must wait for long running

io or guys need you to

cooperative: yield at well defined points, typically only for I/O ops. main problem: use global state. yield. use global state again.
large invarients are easy: can't really do with lockis since suck.

event across i/o: single conceptual task broken into several language procedures. causes a lot of trouble as software evolves.

thread = pre-emptive, automatic stack management.
event = cooperative, manual stack magement.

rip into event handlers
save and restore state across them.

package up a continutation to go from E1 to E2.

when debugging E2 only shows event loop in back trace, not that it came from E1.

manually do things handled by compiler in thread: two or more language functions for one conceptual function. variables once managed on stack must be jammed and pulled out of heap structures. debugging stack sucks.

if foo() changes to blocking, then all functions above it in call graph may have to be changed to pass continuations.
*do an example. make it very clear.

cooperative: view program as one enormous monitor that is released and held at yield (wait()) points.

in cooperative, they make the point that yeilds are dangerous because invisible, unchecked property of function.
what's the problem?

* yield and come back: state changed. have to know that you are leaving the monitor since it will release the lock!

stack management: manual or automatic.
i/o response management
conflict management
data partitioning

cooperative task management = switch when i say to.

big problem: gain in reasoning about concurrency cannot be had without cumbersome manual stack management.

---

general thing about paper:
sosp: 21 papers out of 130 or so.

was a late accept. the vote around the table was 4 against and most people ok with it.

want papers that are really good or really bad. most in the middle.

a lot of things show that they really do use it (the points about GNU libc 2.2 earlier working but 2.3 bypassing syscall stubs, the fact that you can just set env vars to do things).

capriccio args:
 events:
+ inexpensive sync from cooperative multitasking
+ lower overhead for state management (no large stack)
+ better scheduling and information
+ more flexible control flow (not that compelling)

cheap sync: artifact of cooperative.
tradeoff for stack size.

user-threads vs kernel-threads
events vs threads

1: scales to 100,000 threads (does it?)

2. efficient stack (using a compiler to thread together) using linked stacks (ala mesa).

basic idea: check how close to stack overflow, allocate new one if so. don't put in so many checks using compiler.

3. resource aware-scheduling. use control flow to make scheduling decision: is a messy annotation for the set of things you might use next.

this is cute. i like it. but not really compellingly demonstrated. robert will probably have work in his thesis.

2&3 still useful if M:N.

cooperative: boolean vars to do locks.

```
 lock(lock)
```

```
while(lock == 1)
yield();
locked = 1;
```

table 1:
 big difference: kernel vs user.

why create faster than linuxthreads?
slower than nptl?

[i don't understand how a userlevel thread package can possibly be slower than kernel level. should be 10x.]

they say slowdown for stack allocation, but you can easily make this almost free using a good allocator.

they already have a custom one since they talk about doing LRU.

context switching? just saves and restores regs right?
[also goes across the kernel, has to decide what to do next they say reduced kernel crossings and scheduling]]

why the big difference in mutex?
[contended = what? i believe it's = you have to block]

why don't they give us the cost of a contended mutex?

is this the good ordering? what can you do for the first time you switch to a thread? last time you switch out?

---

break into groups: figure out the point of the experiment, whether this is a good idea, and interesting things about the lines.

figure1: what's the point of experiment?
[do something parallel with dependencies and lots of threads and see how things go]

is there a single shared buffer? (i think so).
is the buffer infinite size?

how do you rate limit producers?
[equal number + consumer does work, so will fill up if finite. do you block them all immediately? do they explicitly go to sleep?]

how long can the consumers loop?
just say random.

how is it balanced?
how are the threads interleaved?
[RR?]

[are these with optimizations? i believe so, otherwise its not going to work out at all.]

cooperative: zero overhead essentially. switch out, switch back in, runs full speed. why does it go down after 100? [they say its some sort of cache problem, which is what you usually say when youdon't know what is going on.]

why plausible that its some sort of cache problem? probably does some sort of RR which means that it iterates through all. by t the time it gets to the end it isn't in cache anymore (LRU + cyclic access of data too large = worst case). but if this is so you'd expect further drops.

why does linuxthreads and nptl both start out worse at 1?
why does linux get much worse after 10?
why does nptl get much worse after 100? (100 * 2MB = 200MB. probably not memory problem)

what happens if pre-emptive? lock contention kills things.

if thread holds lock and gets switched out throughput gets hammered. i don't think this can happen to capricio: no pre-empt point between lock acquisition and release. this is another win of cooperative: precisely control when you switch out. some points really suck. various points create scheduling dependencies. always want to switch out with zero if possible.

really bad experimental writing.

why slower at beginning?
[locks]

why slower at end?
[sleep with lock held]

---

figure 2: how does this measure network performance?
[pipes not sockets. on one machine i beleive]

what is the point of mostly idle links? (measures if you go to sleep when nothing, and wake up when something)o

strange: why does it go *up* after the first? (is it over network?)

what does epoll do?
what does poll do?

why does C- suck at first?
[more epoll_wait requests.]
why is nptl better?
[kernel knows already who can run.]

figure 3: overlapping i/o requests.

whta's going to happen to a user level thread package?
[hammered because of blocking]

why does performance go up?
[pick up more requests as you surf around the disk. can sometimes consolidate since there is a lot of locality from triple/double/single blocks+meta]

figure 4: as miss rate increases, what happens to performance?

why does cap suck?
[its aio interface: makes it *2x* slower. this is crazy. trick: check before issue request. since in cache will not need to.

linuxthreads = claim it is a bug. next best thing after caching.

why does everything converge to same?
[can't get any faster, and the overheads must be huge]

---

linked stack:
compiler stack depth: start from routine passed to thread_create and allocate maximum stack. two problems: worst case, allocated all the time, and doesn't handle recursion.

why need checkpoint at all? don't we know the amount of space consumed? (for multiple callers: the graph in figure) when could you elminate checkpoints and just link? (one caller)

basic idea:
put in checkpoints s.t. the stack allocated at C1 is the maximum stack that could be needed by (1) the first checkpoint reached on any path or, if the cfg terminates (2) the leaf.

split callgraph by shooting it with these.

have to do on each recursive.

tradeoff:
check on each call?
+ tightest bound
- high overhead
never check?
+ fastest
- waste space: worst case.

algorithm:
DFS to find backedges.
(basic idea: mark each node,

```
mark_backedge
if(n.mark)
n.backedge = 1;
return;
n.mark = 1;
foreach n.succ
mark_backedge(n.succ);
```

add C at any callsite that is a backedge.o

then: break down further bounds to fit within some threshold.
where to insert?
come in from the leaves, if the current node + longest path > threshold insert checkpoint here.
path terminates by leaf or checkpoint.

will you always fit in bound?
what about code you don't have the source for?
printf may use 8K.

annotations. always switch to a large stack.
if doesn't block that much, then ok.

what about function pointer?

worst case?
- CPU overhead: single call chain

exactly 0 savings. (actually, that's not completely true: have a bit when you return could give to someone else.)

- wasted space: two calls: one big (to make worse case) the other small.

what's guard pages?
usual technique: allocate 8K then mark page after as unreferencable. or several pages (since you could allocate array and then reference too far).

reuse parts of stack across threads in lifo, which helps cache.

can have subpage size chunks (eliminate internal frag)

their experiment for 100,000 threads: call function with 1MB stack that each thread touches. why theirs works well? all threads share the stack, since allocated and deallocated on each call. (cooperative ---- where is the insert?)

apache:
.1% linked stack from C
.5% linked to call external function
10% C determined not needed.
89% unaffected.

"larger path lengths require fewer checkpoints but more stack linking"

---

scheduling:
what runs when. important?
- how far from completion.
- which stages are bottlenecks.

are the points ones that blocked or ones that *could* block? prose kind of confusing. have to open before close, which doesn't match graph. on the other band they are pretty sloppy.

node: blocking point. all blocked threads at one of these nodes.
edge: all subsequent blocking points.

[why differentiate callchain?]

average time for edge
average time at node (weighted average of outoing edges)

average resource usage at edge
average resource usage at node
CPU, memory and file descriptors.

have a selection of things you can run, would want to do the ones that release resources under contention.

preferentially runs tasks close to finishing, since these are the ones that release resources.

* blocking latency: want to shove tasks through as soon as possible since they have to get through a given number of block points.

why "when resource usage is low we want to preferentially schedule nodes that consume that resource, under the assumption that that will increase throughput"?
have to block 10 times: want to start as soon as possible.
maximum freedom: can do it now, better do it now. later might be contended.

they also run if it increases usage when the thing is low. this seems a bit weird.

what would cause all this to break? if average meaningless since tasks do whatever.

they say hard to measure thrashing. seems pretty easy if you look for a jump in the cost of each node. work out low average, then if it goes high, can switch.

---

limit buffer cache to 200MB. why might unlimiting hurt their relative numbers? (compared to pre-empt. which handles pf. which isn't going to happen so much if you have 1.800MB for VM).

served 3.2GB of static content

what is the overhead of this blocking graph stuff?
30% for knot.
8% for apache.

what the hell is going on? why is it so expensive to figure out what edge you care about? why don't you just have a pointer back to the parent edge? don't all things that not block collapse to a single line? [they use the callchain --- i don't see why they don't just haave a pointer back. they should show it matters.]

figure 9: not very compelling. for < 100 clients no big deal. for greater get a bit more speed. but this is for apache, which is really slow.

---

assume we use cooperative threads rather than outerhouts, what changes?
- not preemptive.
- one cpu.
- no sync necessarily: get big critical section except when yield. could put in can_yield().

- no locks so no circular locking dependencies.
- can yield in random places; but in events, could have code that runs after the blocking occurs

how similar to events?
- long running makes

- no concurrency

+ maintain local state

what is the delta between ousterhouts threads and cooperative?
- mesa style: never interrupted
- one cpu.

+ essentially what this transforms to in mesa terms would be running with interrupt enable/disable as concurrency control.

they said bad. what is the reason it works ok?

implications:
- large critical section, except when you yield.
- locking fast (just a var)
- no locking deps.
- unlike events: don't have to manually switch contexts.

what do they say is main problem with cooperative?
implicit yield. we already saw this in lampson. what would this be called there? if you did a wait and all monitors above you released lock.

```
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
*
*  VMS notes
*
*
*
*
*
```

levy: advisor of anderson and bershad (already read a few of their papers). only prof at a reputable school without a phd.

age shows:
- memory is about 125x larger now. (2GB vs 8MB ---
1:250 and this is for
the high end system, so easily 1:1000)
- memory much more sparse.
- pages about 8x larger
- the advertisement is hilarious.

- tlb still about the same size!

- tiny memory (250K to 8MB)
- family of computers: range of 32x in memory size
- slow hardware (one disk) to fast (multiple)
- 32-bit address space (pretty big for then) cliche is that the hardest thing to overcome is insufficient address space size (c.f. x86 seg register hack)

- sort of raid/mirroring idea: have a dedicated swap device. no one does this anymore, would use two disks for more than that (stripe across both)

---

basic layout:

- (similar to mips): 32bit, high 2GB for sys, low 2GB for user.

- OS aliased into each process address space: one pt for it, never switched.

fixed vector at the beginning (poor man's dyanmic linking) used to allow relocation: fix index and the actual location can move around.

- zeroth page not mapped - helps both user and system

- address space supposed to be a few contiguous ranges:
- map with a simple linear array.
- these arrays are mapped with a system page table.
- can be paged out if their mappings are all paged.

- P1 holds stacks for os to use.

- contig physical memory.

---

pte structure (32bits):

```
        30:27  25:21
+-+----+-+-+-----+-------------------------+
|v|prot|m|  os | physical frame number    |
+-+----+-+-+-----+-------------------------+
31 26   20:0
```

valid bit = 0, then 26:0 can be used by os. would typically stuff a disk address in there.

small page size: what is the tradeoff?
+ low internal frag (good for low memory)
+ more precise modification knowlege (less likely have to write)
+ though not really discussed: the minimum fetch unit smaller, which can help.

~ can still cluster and get benefits of larger pages; but i guess its natural to not do so as consistently.

- need more PTEs
- need more tlb entries.

max amount of physical memory? 2^(21+9) = 1GB.

why not put the physical page number at the top of virtual address?

```
   31     9
| vpn  | offset |
```

what problem does this cause if you usually touch stuff near you?

page tables fixed in number:
- each page table defined by two hw registers: [base reg, lenght reg)

three pts in total:
- P0, P1 registers hold virtual addresses ; switched on cswitch.
- system page table.

cswtich: change P1&P0 pt registers, flush user portion of tlb.

---

paging goals:
1. isolation: one crazed process does not hurt the others (that much; "where can it still cause problems?" - turnover.)

how solved?
- isolate each process has a resident set + resident set limit.

- when you reach your limit, every request causes you to give up a page.

- when there isn't enough memory, an entire process gets swapped out.

argues: allows quick decision (take off head, but do have to search).

*nice feature: if system not under memory pressure then 0 overhead: nothing gets put on or taken off.

the free list: how long stays on a function of list size + fault rate.

what if resident set limit too high?
don't reclaim pages, sit there dead, isolated.

too low?
reclaim pages often, will not have any performance isolation. have cost of movement.

as freelist + modified list goes to
0 replacement goes to FIFO
size of memory goes to LRU! why bad? (turnover)

what is the overhead?
200usec per reclaimation
10ms if you take and flush something.

can do 50 mistakes of moving page back and forth before equals the cost of a page fault.

as you make list larger, minimize chance of pfs, but increase the number of faults + recoveries.

how does it compared to clock?
- you have to reference between the time you put it on the list and it gets to the head.

- clock: you have to reference before you get through all memory

if only one process: one long fifo stream, but do lru once it gets kicked out. in a sense, they want non-locality: once on list will not reference that much, so lru not expensive. the list actively being used, will be referenced all the time, so don't want to do lru. cute.

- i think they are essentially the same if the freelist was the size of memory.

not sure what happens to modified pages after written: put at front of freelist or end? survives longer if latter. weird discontinuities.

freelist + resident set of one process acts pretty close to two-handed clock. bit more complex with multiple ones though.

tradeoff:
- each mistake:
    best case: list insertion, search, removal (200 usec)
    worst: flushed page = ~10ms.

- correct decision: saves a page fault (10ms)

some set of processes make nominal progress: swapper pushes out processes that don't fit so that highest priority stay resident (not clear if priority fluctuates with eviction as in unix) not loaded unless enough physical memory for whole resident set.
this helps reduce disk workload (less paging) and sort of limits the effects of whacked out processes.

2. demand paging cost

- they cluster pages together and read them in en masse (how large cluster?)

- similar: swap in entire resident set.

3. increased disk workload

- minimize i/o's by writing things out in large contiguous chunks.

- also with the fetching (reduces seeks & requests, but does cost more in terms of bringing in memory)

- also by deferring writes can (1) absorb more and (2) eliminate them if the process exits.

seems to have scatter/gather io since no discussion of allocating in contig physical memory (makes sense since mainframes had really sophisticated i/o systems: reasonable that this part gets down --- design is free in a sense)

modify list: have a low-water mark & high-water. when reaches high, blast out (high-low) pages. "why not write out the entire list?" could have been stuff you just freed. means no caching -- big spikes in how things perform. two different time scales: how long been on, when you flush. better to seperate

modify list has two thresholds:
high limit:
too high?
bias against unmodified data
too low?
write out too few.
low limit:
too high?
write out too few (so reality: it's the delta)
too low?
no enough caching going on. big spikes.

high-low too small?: (above)
high-low too big? (tie up disk for a long time; massive dip in performance)

good thing about deferred writes:
1. absorb modifications
2. cluster contiguous pages together
3. write a bunch at once
4. can return back to user without any work (if modified likely modified again in the future)

(1) & (2) & (3) all the same in hardware cache (why you have cache blocks rather than bytes)

essentially have two-level paging: one at the granularity of pages, the other at the granularity of processes.

4. the processor time consumed by searching page lists

- don't use reference bits (morally the same)
- use fifo + second chance: no search, except on page fault (do they use a hash table?)

---

pager runs on page fault. can get from:
1. swap file
2. a.out
3. demand 0
4. freelist, modified list
5. a mapped in file (theyseem to support mmap)

OS is paged: difficulty?
- fetch and eviction code can never be paged. neither can anything they call (printf, find-first-bit, linked list routines, device drivers, timer code, lock code, ...)

segments: link everything for non-paged, put contig, then link all pagable. as long as you get the roots right you'll be ok.

- hold a lock: have to be very careful with load/stores. at the least you hold it for milliseconds. if you have a big kernel lock have no protection.

---

basic theme: large fixed cost to moving head + incrmeental cost of writes. many file systems do the same thing (LFS, hardware caches to (don't fetch byte, rather get more)

all the places they do contig:
- a.out: linker lays out, if not successive, then it goes backwards (i think)

- swapping (lay everything out contig & rewrite if any io in progress)

- paging: writing out dirty pages is deferred and they are ordered before

batching: when faulting in, when writing out.

---

missing things:
- no discussion of defrag.

- it's all english text: no formulas, real algorithms, etc. systems is a bunch of observations. no answer to: what is the best limit? what is the best size?

- how do they pick which process to swap out? swap in?

---

discussion queestions:
1. how would you actually design the experiments?
2. what is a worst-case (or very bad) workload for this system. (compared to what though?)

- poor spatial locatlity so that clustering does the wrong thing

- fragmented disk so that you can't lay things out nicely.

- bad for all paging systems: bad locality in general, one reference per page.

- reference one page per group covered by a page table page (512/4 bytes = 128 pages per PT page)

- ah: each process has 10MB, one needs 11MB, one needs 9MB>. performance will really suck.

---

experiment:
- saved 50 page faults (about 50*10ms = 500ms = half a second)
- no end to end performance
- 359 pages were sufficient for 4million references (1:10000 ratio) [total size = 183K]

- gets better and better so why not make freelist be even larger? 200usec overhead + non-isolation.

way to look at this: one long FIFO list, but you do LRU on the free list segment of it. locality means less references to it, so few operations.

really want to see the end-to-end time, plus the cost of doing LRU.

experiment table 1:
- half the faults handled.
- no idea how much memory
- 70x more read ops than writes (since can batch)
- read to write ratio about 4:1

- faults from modify and freelist sort of equal --- why? does this mean they are the same size? does modify last longer?

- total page faults is about 80K

- very few write operations compared to read: 7K to 117. usually wrote out about 100 pages at once (.5 MB)

- total page faults should be the sum of
pages read + faults from free list + faults from modify list
+ demand 0

but its not. get 79356.

* how would you hide things?
    - tune until works
    - drop things that don't look good.

****
****
****
****
*
*
* Superpage notes
*
*
*
*
*

first example of a modern experimental evaluation. varied every possible parameter. typical of peter: solid, careful papers. not revolutionary but interesting and you can stand on them. if you were going to build a superpage system you would read this paper. just as with the mesa paper, or the eraser paper.

very common type of OS paper: some pretty simple idea, goodness measured in terms of performance, huge amount of measurement. typical improvements by community: arch (1-3%), compilers (5%), OS (20% or more).

notice non-trend: still mostly english text, no formulas. just measure more. so vms in this sense isn't much worse.

grail: achieve high and sustained performance for real workloads and negligible degration in pathological cases.
evaluated on real benchmarks
built in real system

funny case where hw evolves faster than sw: superpages out for over a decade, but very little support in os. primary maps framebuffer using one big superpage (maybe OS as well).

paper obsolete if hardware can (1) make big TLBs or (2) form superpages out of non-contig pages efficiently [fang et al]. if past=future then they won't do (1) anytime soon. not sure about (2).

+ fundamental assumption: do no harm. should always be better, even by sacrificing upside by taking less risks. where can be worse? (1) general overhead, but they show not a problem; (2) paging decisions. this is on place where it does seem that they could be screwed.

a huge amount of care is given to screw cases. you are almost certain that if you use this, it will not blow up on some workload out in the lab. have no such warm fuzzies about the vms stuff.

---

thing to keep in mind: just mapping integers to integers.
vpn -> ppn

most of the problems in the paper boil down to the fact that there is no good way in general to construct an arbitrary int->int function. if we want to map arbitrary ints to ints, then we fundamentally have to use a brute force table. ie.., O(n) space

quick OS proof: if its purely random, then its not compressible and we need to record everything.

so the way we handle it is to induce structure (restrict flexibility) so that we don't have to specify. the way we always do in OS is to force consequtive vpns to go to consequtive ppns. segmentation, pages (larger and larger more of this), superpage.

---

versus malloc and free

similar degrees of freedom:

1. where to place: step through whre to place things?
2. when to coalese: can form back into big one.
3. what to split: which space do we break down.

differences:
4. (different from malloc): when to move.
5. which physical you free
6. units very much larger.

similar lack of control over:
1. size

2. request stream order (not completely true: can suspend program).

3. the virtual addresses (mostly: can chose mmap, or the initial place to map stack and such)

4. which/when virtual you free.

placement:
    worst place to put one block?
        [leave largest free that you can: they sort of do this using best fit strategy]

    populations of two different sizes?
        [split on ends: could segregate memory [don't do]]

    populations off a bunch of different sizes with correlated deaths?
        [cluster together: all pages from same process [don't do]]
        [their wired pages, seperate out: these cannot move at all]

if you look at it as similar to malloc and free you notice a bunch of heuristics developed there that could work here. you want to put things of same size in same place (all same size no frag); you want to put things that die at same time in sample place (e.g., bias alloc of all stuff for same program in same place). they don't do either.

in general: large unit allows us to satisfy the largest number of different. don't split when a smaller one will do. try to free up big chunks at once. in trouble when only small.

Do we care about the problem?

Figure 1 is our tragedy: log graph, with suckiness increasing exponentially (100x over 10 years). this the the wrong end of moore's law. overheads of 30-60%.

- physically indexed cache: what is the problem? cache *hits* can cause TLB *misses*. wild.

- so use a virtual cache. what sucks? (flush, aliasing, consistency) solve aliasing with hw or by forcing sw to map so it conflicts. solve flush with asid.

- could have a two level TLB, but i don't know of anyone that does this. given how much space wasted on BTBs, not clear you wouldn't want.

- plausible worst case access pattern? 1 reference per page. which benchmark looks something like this?

mips? 64 * 4K = 256K! tiny.

our simple solution to make the TLB go farther: just increase the page size. problem is internal frag. so we allow increase of page size across a set of ranges (fit it). sort of like segmentation, but much more restricted.

Alpha:
two tlbs, one instruction, one data. (128I+128D)*8K = 2MB. in their case 500MB, so factor of 250. since multiGB. easily factors of 1000s.

(why put vpn at the top of the virtual address?)

8K, 64K, 512K, 4MB, where va % 64K = 0 for 64K superpage, etc. why is it aligned? [so that you can steal bits for the size without needing a larger tlb.]

good:
+ larger coverage
bad:
    artifacts:
- one ref bit
- one protection bit
- one modify bit
- aligned both at va and pa.

    intrinsic:
- one reference makes whole thing refereneced. bad positive correlation: larger it is, more likely it is to happen, more memory wasted.

- one write makes whole thing look modified. bad positive correlation: larger it is, more likely to happen, more costly to write to disk.

- hard to allocate in a good way.

- costs more to write to disk (if you were more precise not bad?)
- less locality: pr that you use byte b0 and b1 decreases with distance.

+ how much does a TLB miss cost?
+ how many do we have per page fault?

---

allocation (triggered on page fault)
- best: all of memory free
- worst: only 1 page free or scattered.
[anytime contiguity important this is true: identical on disk]

basic tradeoff: greedy now or take long view. in both cases don't know future so may be wrong.

- local view: want to put where it can grow the longest
- global view: don't want to waste

their heuristics:
1. leave as much free space as possible. (take from smallest part)
2. don't preclude future.
3. don't be worse than nothing.

how much to allocate?

- if we knew future: would make exactly as big as needed.

allocates exactly one base page, with the same alignment as the faulted page, but reserve the following pages. reservation size is the largest superpage that is completely contained by object. will promote to superpage when fully populated. in the meantime have a non-binding reservation. free to take away, but will do so in FIFO order.

how do you use up a reservation? promotes gradually: as soon as extent fully populated goes to the page. this is an example of the do no harm; works well in practice since often if you're going to use, do so promptly (e.g., an array initialization).

in case of alpha: promotion = replicating exact same PTE to all relevent PTEs. TLB miss on any will bring in the same entry.

what to do if doesn't fit? where does the reservation come from?

1. freelist.

2. if cannot get a reservation, preempt an existing one. sort reservation list by when you did a page allocation (i.e., allocate page, put at end): LRU = take from head, O(1).

3. if no more space: could copy, could flush things out; they trigger the coalescing deamon. (will talk about later).

rellocation:
1. can do because you know where pointer to relocated object (page) is and so can update them (i.e., modify page table). cannot do this with C & malloc since you don't know pointers.

2. rellocation always interesting when you care about continuity. usually maps to some sort of mark and sweep garbage collector. called different things: defragmentation (disk), garbage collection (memory), continuity recovery (superpages).

to steal some good ideas from malloc:
1. want to allocate things that will die at same time together: can allocate objects from same process next to each other.

2. want to allocate things that are the same size together (since no frag if die & allocate more of this size in future). could possibly do histogram computation to determine how big zone should be.

3. allocate things of same type together: (1) wired (they do), (2) file blocks (survive longer than process).

can either select from these in order, or use recursively.

---

promote: want to grow it: how much and when?

- too early, will eat memory for nothing and someone else might have made better use.

- too late and something else might be there

+ does when entire range is populated.

if you need 56K do they allocate 64K?
if you need 4MB - 8K do they allocate 4MB?

on alpha:

a fully populated 512K region that goes to superpage will have been populated by 7 64K superpages first.

observe: populate densely and early.

demotion: (evication) exit or lose page:
1. occurs on eviction. recursively breaks down to largest smaller superpages.

2. under memory pressure: when deamon resets reference bit, demotes superpage to base pages and repromotes back only when all referenced. (anything else?)

3. first write to a clean supepage shatters it. only have one reference bit. so don't know what parts are in use, so break down into a base page that holds the dirty one, and as large super pages as possible for the rest.

4. change permissions on subsuperpage.

---

they constantly avoid doing worse than nothing. where could they do worse than a system without superpages?
- overhead (but doesn't seem so high)
- page eviction decisions: demote to get continuity, demote closed file pages.

---

how does evication work?
freebsd:
[- four lists: free, cache, clean, unmapped]
four lists:
- free (not said) pages that correspond to nothing.
- cache: clean and unmapped (file data).
- inactive: mapped into process but not referenced in a while (dirty or clean)
- active: accessed receltly but may not have ref bit set

mem pressure:
- moves clean inactive -> cache,
- pages out dirty inactive
  - deactivates unreferenced pages from active list

modification: all clean pages backed by file moved to inactive list as soon as all processes close file. these will only be picked up by the coalescing deamon.

reservations use both cache and free pages. take free over cache.

deamon actived both under memory pressure and when continuity low. (allocation fails: deamon walks over inactive list, moving to cache pages that will make continuity to satisfy request! don't move if

don't help. stops when run out of list or made enough for all past requests. do no harm.)

downside: reused sooner than lru since contig.

---

experiments:
- best case, under stress, pathological.

+ huge set of standard applications, so you can't hide weaknesses.
+ very precise set of attacks on the system.

- real in that it runs on an actual system, but only done on that one system. you'd expect things to work out well on others, but no real demonstration.

- missing numbers: what is the cost of a miss??
- what are the benchmark times in seconds? cannot figure out if it really matters without this.

6.3: best case: free memory plentifuel nonfrag, every attempt to allocate succeeds and reservations not preempted

"is it worth doing at all?"

- mean elapsed time with warmup. why warmup? [want to remove I/O effects -- could overstate superpage contribution.]

- many more base pages, but coverage mostly comes from large ones. (correlation in ratio to speedup?)
512 8K pages = 1 4MB. so often have a factor of 80 or so covered by the large page.

- why mesa slowdown? doesn't track zeroed out pages

- web has few big pages (allocates many small files so doesn't benefit)

- fftw has large number of huge pages (almost no speedup with < 4MB)

- matrix goes crazy: misses in tlb all over place: how one miss per two accesses?

- the page coloring thing is a really good example of how careful they are. could have easily turned this (witout realizing it) into a page coloring paper. page color intuition: assume spatial locality. make sure that close together thigns do not conflict in cache. 64K VA will map to 64K PA range in low order bits.

- big problem: they never show runtimes. 70% in a short program probably don't care.

6.4 multiple page sizes: only allow single allcoation and compare 8K (base) + 1 superpage size.

- 13 out of 35 that had diff (the others didn't!)
- looks like 512KB gets ms of the benefit!
- doesn't measure space consumption so not full picture
- mcf doubles speed up when can chose between sizes
- fftw makes big diff. why?

- big problem:

their significant digits are nonsense. alpha only counts every 512K misses! so for short programs this can be n or n-1 easily, which makes a *2 digit* difference

6.5 handling frag

- run verous programs (grep, gcc, netscape) and within 15 minutes have sever frag: no region larger than 64K even though 360MB out of 512MB was free

- restore continuity using the deamon scheme

- cache coalese all cache pages (clean unmapped) - deamon (trys to build up larger) by moving inactive to cache (coalescing) andy by doing wired page segregation. key: treats closed files as inactive as opposed to old version which does not demote immediately from active->inactive. if no memory pressure can remain active!

measure impact: run webserver after this experiment (i.e., when evictions have occured). slowdown = 1.6%

fig 5 interesting:
- lose continuity over time
- recovers, then runs fftw, which consumes, then releases recovers, consumes releases (each spike gets higher)

- ignores inactive pages: could get *more* continuity by moving them, but only do so on request, so passively seem like less contig there.

why doesn't recover all if its available? [does not relocate. so may have enough in aggregate, but allocated pages sitting in middle. need turnover in these.]

can only recover 20 out of 60 requests. weird: only gets 35, out of 60 but this gives = best case speed up?

6.6. worst case:

1. cost of promotion: allocate memory, touch one byte per page

dealloc (overhead is all promotion in sequential pattern with no reuse). overhead is 8.9%

why 3 incremental promotions?
8K->64K, 64K->512K, 512K->4MB.
they say allocate "some memory" so must be > 4MB.

if you memcmp, then overhead is .1%

2. preemption overhead: preempt all the time, 1.1%. not clear about this

given: 4MB contig.
1. reserve 1 page 4MB.
2. reserve another +4MB. will break 4MB into 8 512K regions. the first has a reservation. the latter will have another.
3. do six more allocations.
4. all 512 reserved.

will become 64 64K regions.

3. overhead in practice? very nice experiment: do all benchmarks, but don't actually do the promotion (still have to do allocation defrag etc)

4. big benefit from dirty demotion.

****
****
****
****
*
* ESX Notes.
*
*
*
*

non techniqcal:
- carl has won 2 out of the 5 best papers at OSDI.
- libertarian that decided to not go academics since he didn't want to take government money.

- real system.
- commercial. good commercial papers are rare (vms has been the only other one so far).

acording to mendel two main things esx used for a lot.:

1. run N app on NT/windows = crash. run one app on NT /windows = no crash.

Note: nice way to trade memory+cpu cycles for reliability.

2. security isolation: hard to break into vmware. only compromise guest os.

how to run OS as an application on top?
1. have to intercept all input from the environment.
2. intercept all places where it interacts directly with env, or save/restore environment on switch (e.g., registers)

3. paper's main argument: statistical multiplexing. machine mostly idle. rathr than have deadicated one, throw a bunch of the same machine and just buy a faster one (or save the money). if not mostly idle then this does not work that well. though: in the case of memory, could make an argument that aggregation helps a lot.

four main tricks:
1. how to get pages from the OS.
2. how to share pages across OSes.
3. how to integrate proportional share with some system feedback.
4. i/o pages.

---

abstractly what is virtualization?
- insert a translation point between all uses of physical hardware and map them to logical uses

- typically used to go from a small, fixed number of physical resources to a near infinite number that as we exceed just gets smaler.

- virtualization = translation. intercept + remap to what we want.
    for speed, in reality: get a trap on all important events
        - use of privileged instruction
        - kernel address

...

can always just interpret the whole thing and force it to do whatever we want.

when they say non-virtualizable, really means what features prevent us from running it directly and just catching when it does something bad with a trap.

if i think i have X, do i?
- register r1? yes.
- r/w tlb? sort of gets relabeled.

- PPN? no.

what isn't virtualized?
    registers: save and restore
    tlb: flush

---

what does memory virtualization mean? just one more layer of indirection VA -> PPN -> MPN.

we've already done this. random association. need a table. PT for VPN->MPN. what for other? [pmap]

easier for PNN -> MPN than for VA->PPN?
dense, small, just use an array.

how to insert this layer?
- every tlb insertion have to relabel.

- every address they give to dma device have to redo. and if the page is too high, have to copy it.

- if they read the tlb have to intercept.

- if they can write to raw physical memory have to intercept loads and stores (e.g., on the mips)

- when you switch between OSes, may have to pin additional tlb mappings

- NASTY: if they relie on conflict misses to synchronize the cache you may be in big trouble. easily.

what can't we hide?
time.

overhead:
32MB per VM, 4-8MB used for framebuffer.

---

page replacement:
Perfect practical = LRU. Perfect worse with ESX = LRU.

if we take our own LRU page what is the problem? if we are roughly synchronized with guest os and both make similar decisions, could both decide to page out the same page.

if we had good knowlege: best thing would be to take somethin off the guest freelist.

how to find the guest freelist? call getpage() (or equivalent)!

another possibility would be to give the guest OS a huge amound of physical memory, so that it is doing eviction paging relatively less often. what is wrong with that? (one thing is that most of its evicted pages will in fact be paged out over time so your guaranteed that they won't be in memory) also we will blow a lot of overhead in the guest data structures.

one change in view:
naive: only emulate machine to trick OS. not allowed to do anything else.

refined: view OS as a black box. can use any well defined interface (syscall, driver interface, machine).

figure 2:
black bars: configure OS with exactly that memory.
grey: configure with 256MB, balloon down to the size.

1. why does throughput go up with size?
[large working set: can use more cache]
2. why is grey bar smaller than black?
[overhead of having more mem]
3. why overhead larger with small?
[same configure, balloon down more = more overhead]

---

sharing memory

running multiple copies of same OS: want to coalesce duplicate pages.

how do i decide to share?
-scan or at page out.
- hash page.
- look up in hash table.
- compare equal.
- if already entry:
- if shared? just increment refcnt.
- if not shared before?
mark COW. [what does this mean?]
change entry, refcnt=2
- if not equal but hash same: do nothing.

- if not there
insert. what else?

table indexed by hash of each page. maps to either
hint frame:
hash, PPN, MPN, the VM that has it (so you can go change page table)

generated by a demon that hashes. mark the page as COW after hash? no. first write would screw it up.

do a random scan; also always attempt to share page before pushing it out to disk.

this is why hint: not actually true. may be false.

use PPN:VM to go find and modify the page table to change to read permissions for COW.

shared frame:
hash, MPN, refs.

mark page as COW. big overhead? allocate zero filled, then write to them.

how to demote? doesn't seem to.

nice about random? will bias towards long lived pages! spagetti paradox: pull spagetti out of pot, will get longer strands.

two experiments, just like superpage: "[sort of] best case" and "typical case"
missing: experiment that measures real amount of memory free

Figure 4;
run 1-10 linux OSes with 40MB spec95 benchmarks for 30 mintues.
same thing, but probably not that well syncronized, and small buffer cache means won't keep around.

[why granularity different? 30 minutes rather than sec? i think because its for servers so just care about asymptotic. do care if it can handle bursts though]

different between shared and reclaimed?

- the memory actually taken back. if two people share a page, shared = 2, reclaimed = 1. if 10 people share, shared = 10, reclaimed = 9.

  so the degree of difference inverse proportional to sharing: more people sharing same page, then goes to diff of 0%.

- gap at the beginning means that less people share than later on? why is that? of course: less vms. then get more asympotic.

- small at beginning then spike because 55% of sharing

with 1 vmm is the zero page, which will have many many references.

- will get 1 for each shared page, so is essentially a count of the number of shared pages. would be better to have a line "ave ref count" and "number of pages shared"

- why doesn't shared line start at origin? why sharp initial spike? why tail off quickly?

- why does it only go up to 67% shared rather than 100%?

- say they reran w/o sharing and didn't perform worse. can they draw this conclusion [if there is no paging, reasonable.] how big is machine? is there paging?

Figure 5: "typical"
- why less sharing as you go down?
[could be that linux is tight with less dup [most dup in os itself] could also be that it sucks at zero page management: 25MB of 120MB saved for linux, 70MB of 345MB for windows]]

- can view zero page as an annotation that don't need.

- about 1/5 due to zero pages for last two.

32MB each VMM, 512 -> 360 available, so 320 + 120 = 440 used for virtualization, which isn't an impressive win. (only better for A)

---

shares versus working sets.

want to partition for similar reasons to vms, but not get penalized by idle parts (as much).

need to incorporate feedback from the system. e.g., vms just had the resident set limit, but didn't have any way to adjust even if the other guys were not using.

want to know how much of memory is idle.
samples 100 pages every 30 seconds -- what happesn if this sampling period goes to zero? [actually don't want fine grained measurements! cpu bound app, or blocked on i/o]

use 33 pages after 30 seconds: claims 67% of memory idle.

have p1 and p2, need memory: how?
min(s1/p1, s2/p2): many shares, few pages -> more likely to keep.

fuse feedback with proportionality:

$$p = \frac{S}{P\,(f + k\,(1 - f))}$$

k = 1 / (1 - r), 1 <= 1 < inf
r = taxation rate, 0 <= r < 1

f = amount of active memory. put a tax on idle memory (1-f).
75% tax means that will take back at most 75% of idle mem.

how to get VMS? need (f + k (1 - f)) = 1.
r = 0 implies k = 1
(f + k (1 - f)) = (f + 1 (1 - f)) = 1.
or f = 1
(f + k (1 - f)) = (1 + k * 0) = 1.

if taxation is 100? k = inf
P (f + inf (1 - f)) = inf
but it's relative, so washes out? really weighs any f < 100 very strongly.

if f = 0, then scales in direct proprotion.
P * k
what does curve look like as a function of usage?

[*]draw (f + k (1 - f)) for a bunch of ks.

[magnitude of it?
0 <= f <= 1
1 <= k < inf

f = 0 means
f + k (1 - f) => 0 + k - 0 so directly tracks k.

f = 1 means tracks f.
min(f.min, k.min) <= . <= max(f.max, k.max)

0 <= ... < inf]

three averages:
1. slow moving
2. fast moving adapts to working sets quickly
3. super fast

esx uses maximum to estimate amount of memory used. implications
1. won't take away that quickly.
2. will give credit quickly.

figure 6:
why do estimates trail?
  on up:
how slow does it trail? (looks like a couple of minutes)
max works pretty well, tracks it very closely.

  on down:
both trail by about 1.5 minutes.

- why above?
does it work?

figure 7:
- huge overhead: out of 512MB, only 360MB is available for users!

  if we have a .75 tax rate, and 0% use of memory, what is the fraction
  we get?
S / P * (0 + 1/.25) = S/4P
S1 = S2, P1 = P2, then 1/4

... and 100% use of memory?

S / P * (1 + 4 * 0) = S/P.

increases amount of memory used by 4x. could have
allocated 4x more memory and used 100% of the time for
the same share.

why doesn't the line go up to
180 + 180 * .75 = 315?

[max configured: 256. min is probably 100, though doesn't say]

we really can't figure out how well it works. you'd need to
configure max to be 320 at least.

why does VM2's line go up at first? booting, so pretty busy
zeroing pages (windows) drops after.

---

allocation policies.

how much memory a guest gets is determined by:
min: guaranteed, never take.
max : configured linux to think it has that much.
shares: relative proportion of memory you can use. 2x = 2x more mem

- max = min, then shares don't matter.
- if not overcommitted then you get max.
- if sum of min + overhead = the amount of physical memory,
then do not let other VMMs run.

- how much aggregate disk space?
sum of (max - min)
- will never page you below min. ever. does this cause problems?

when does page deamon kick in?

do nothing
high (6%) --> ----
start using balooning (paging if no baloon driver)
soft (4%) --> ----
paging
hard (2%) --> ----
paging + block vms that > min.
low (1%) --> ----

tries to always be above high. the system transitions to a higher state
only after significantly exceeding its threshold. funny that it is so
close to the wall. disk latencies are so slow...

figure 8:
1. windows exchange benchmark (2 vms, min=160MB,max=256MB mem)
a. exchange server, windows 2000 server.
b. load generator (to a), windows 2000 professional

2. citrix metaframebenchmark (2 vms, min=160MB,max=320MB mem)
a. metaframe server, windows 2000 advanced server.
b. client load generator, windows 2000 server

3. sql server (1 vm, min=160MB,max=320MB mem), windows 200 server.

Sum = 1.4GB, machine has 1GB
- share before swap:
- 325MB of zero pages not sent to disk.
- 35MB of non-zero written to disk.

  - why initial dip? [zeroing] page 32MB out of 325MB actually hit
    disk because of share before swap.

  (d) idle so gos down to 160MB (min) or so. query kicks in, then
  jumps back up. (at the same time, balloon goes down since
  memory is not idle)

- what determines the size of the spike up? 320MB is hte max (check)

(b)

- why does sharing go down? (run different apps. at the beginning
  lots of zeros, and shared code)

- as active shoots down, why does alloc not go much below?
[configured with 1GB: always try to get that much in
  use]

- between 20-30 why does it go down much more than 1GB?
[overshoots when transition to hard]

- why does it go above 1GB? might be accounting? or includes
  vmware mem too? i think it counts shared memory = S*refs.
[aggregate allocation = 1.2GB]

- why does it mirror the ballooning line?
[taken back with ballooning, mirrors active too]

- why does ballooning mirror the active line? [different apps.
one goes up, the other app gets ballooned]

- how much shared? (about 20%)

- two inital peaks, one pushes to the low state, one pushes to the
  hard state: why? [i think is partly because (1) booting so cannot
  use ballooning and (2) first peak is a rapid increase from almost
  nothing, the second is a relatively small delta + already over
  shot goal]

  (c)
- active tracks alloc morphology really well, as ballooning does
  inverse

- shared rises over time, any ideas why?

  the fact that balloon mirrors alloc must mean that the guest os
  is pushing, trying to consume more memory.

  interesting: seems like taxation kicked in, since active in citrix
  causes balloon to go down, and alloc to go up in a very similar
  fingerprint pattern.

---

how different than vms?
- get OS on top to do some of the work (ballooning)
- fluctuates the RSL
- how different than LRU?

- how many references per page on average?
67% of memory shared.
60% of all memory is reclaimed.
1 page for each shared group.

  assume 100 pages.
67 pages shared
60 reclaimed.
7 pages left, so 7 distince groups.

reclaim = (unique pages * (references - 1))

reclaim
    ------ + 1 = refs
    unique

reclaim
    -----   + 1 = refs
    shared - reclaim

60   + 1
    _   =
    7

9.57 = refs.

makes sense: have 10 OSes.

  why not exactly 10? (probably some inter sharing, plus any stuff
  that gets not synchronized)

  refs for other boxes tables
A = 673/(880-673) + 1 = 4.25
B = 345 / (539-345) + 1 = 2.77
C = 120 / (165-120.0) = 2.67

isolation and performance are conflicting goals. fundamentally
the only thing you can do is decide how much to weigh one or the
other. that is it. his taxation does this simply in a smooth
function without weird discontinuities.

```
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
*
* Nooks notes
*
*
*
*
*
*
```

where did this come from?

there was a glut of os extension work in the mid nineties.

straightfoward performance story: kernel does htings in slow,
general way. if can customize get 10x.

this got a lot of superstars tenure (bershad, kaashoek) and got
me a stanford job, stefan (in part) a stanford offer, gun sirer
a cornel position.

but in the end, nothing much changed.

there was a lot of techniques developed to jam code you didn't
trust in vulnerable places.

these tricks have appeared after in different contexts.
this is one: problem is easier.

how do they sell their trick?
drivers account for majority of os crashes.

how to fix?
better testing? better language? better IQ?

simple:
1. catch driver error
2. free its resource (must track them)
3. reboot it.

 What are three goals of the system:
  Isolation - don't let fault in one extension infect rest of system
  Recovery - support automatic recovery after a function
  Backwards compatibility - e.g., work with Linux

demands:
- detection: no detect, no do.
- reboot must fix error. not determistic.
- what happens to application?

zero-modification backcompat as a big thing.
what is the alternative? just write things in a type safe
language. lots of dead systems that did this.

make sure to keep hindsight test in: if someone told you they
could get rid of 85% of OS crashes without modifying anything,
you'd think they were nuts.

deployment story: if you just want to prevent, is pretty easy
to do at the level they did. they want it to be transparent.

how compare to alternative?
- cap = hw, so they are safe
- microkernel = reorganize entire os, so safe
- language = rewrite so safe
- driver arch (same as micro)
  also no recovery for previous

- transactions: don't fit most things. still have to detect
- virtual machine: this is kind of weak.
- static analysis: going to miss things.

what's downside of nooks?
- can be expensive
- doesn't catch everything
- doesn't really work for kernel

says "major feature" of nooks is "virtualizing only the interface between
kernel and extension" as opposed to virtual machines. what does this
mean?

---

nooks = two nouns
lightweight protection domain
XRPC

 lightweight kernel protection domain
isn't this just a misspelling of VM? what's the diff?
just protection; in same addr space.
OS = RO.

how to isolate?
RO kernel.
WR extension.
any kernel data structures you need to write are explicitly marked
- either copied.
- or an id is stored.
switch page tables + stack on call,

 Use paging hardware to protect kernel & extensions against bad extensions
  See Fig 3: Kernel can write everything, extensions only write themselves
 Use Extension Procedure Call (XPC)
 What do you have to do to call into an extension? (Fig. 4)
  Copy any argument data structures to where extension can write them
why copy? what would eliminate?

[so extension cannot trash the data structure]
i.e., protection.

what hardware support would obviate copying?
if you could protect byteranges. this is what capability
systems do. can take arbitrary bytes and associate a
capability with it. each process has a list of caps.

if you have dthe cap you can write, otherwise no.

bill dally built one of these. M machine. these tend to
 not get used all that much.

what is a problem of copy?
- cost
- if someone else modified, last writer wins, doens't
  seem to synchronize

  Might need to follow/adjust any pointers in data structures
what if you don't? extension may crash for no good reason.
you must copy anything they do writes to.

  Adjust stack pointer
why not use the same stack?
corrupt?
  Load %cr3 with address space of extension
  === run extension
  Switch %cr3 and stack back
  Copy results back; synchronize any modified structures
 What about modifications to non-argument kernel data structures
  Fortunately, often done through macros and inline functions
  Can change these into XPCs
 Where do page tables come from when loading %cr3?
  Nooks has to maintain a set of "shadow" page tables
   Just change code where linux touches page tables
   Have to modify page fault handler... how?
  Task structure on kernel stack?
  Could you optimize this process on the x86?
   If extensions are in different 4 MB regions... maybe re-use page tables
   (Just clear PTE_W in page directory entry)
   Or at least do this for some regions (might not work for buffer cache)
   Also, maybe targeted TLB flush in stead of %cr3 load?
 What is deferred XPC mechanism? Where/why does this come in?

what happens when linux modifies the kernel page table?
have to propagate to *ALL* extension PTs?

what would we have to do if we wanted to protect each procedure call?
entire OS would essentially be RO.
copy in its parameters.
copy them out.

if it blows up? this is kind of the problem. must be able to
restart it. not really clear how to restart qsort. if you reset
its state and go forward will blow up again. could blow it up and
go back up callchain until you hit something you could restart.

why not do for every procedure call?
huge performance kill. multics did this actually. never

made people that happy.

probably faster to do a better language. or recompile C with
a safe C compiler.

a problem with asymetric.

if we could fix C to provide the protection they need, what would we do?
the only protection they give is that you cannot trash pointers
or write beyond the end of an object. i believe. this is straight
mem protection.

actually i think they do some parameter validation. at least
for lifetime.

What are wrappers? How do they work?
 Three purposes:
  Check parameters for validity
  Implement call-by-value-result? (What's this vs. call by reference?)
  Perform XPC
 Basically works through linker
 Who writes a wrapper?
  Tool auto-generates skeleton from header
  Fill in by hand
 Need to know properties (Perhaps extractable by Metal)
 How specific to each extension is the wrapper? See Fig. 5

What is Object Tracking and why?
 Records address/type of all objects in use by an extension
  If used for call, just attach to stack
  If held, keep in per-extension hash table
 If ext. might write object, keep association between kern & ext. versions
 How do you know lifetime of objects?
  By hand inspection - determine type of object
   passed in for call, allocated/deallocated by ext., special (timer), ...
 Do you always copy objects?
  No... more efficient just to re-map network & disk buffers

 problem with copying if other people use?
lost updates. locking does not help.

How do you detect a fault?
 Easy cases... page fault or other exception in extension
 What about harder cases... e.g., no network packets received
  User can detect and initiate recovery

- eat to much resource
- invalid parameters
- exception.
- user can say

How do you recover from a fault?
- Disable any interrupts vectored to the extension, if driver
(what if you didn't do this... could get livelock or worse)
- Invoke user-mode recovery agent
Perform extension-specific recovery, notify sysadmin,
Change configuration, disable after repeated failures, ...
By default, unloads and re-loads module
What's this about interruptable vs. non-interruptible state?
What about allocate memory? (This is why we need object tracking)
What about thinks like network buffers w. pending DMA?
Only free buffers after re-loading driver
after it has re-initialized the device

probelm with release?
things outside extension may be using
kernel may have on list
HW may have in device DMA.

what are their speed hacks?
deferred call. batching: crossing line is expensive. so batch
it up. total hack.

shadow copy of data. do write buffering and then flush. saw
already in VMS. instead of disk, mem, is extension and kernel.

what does nooks not protect?
screw up device, deliver bad packets, don't send.

Set %esp to something bad and take an exception (what happens on x86)
DMA to physical memory you can't write
move something to %cr3
disable interrupts and loop forever
logic bugs that don't involve trashing memory

why not protect against pt register modification?
does it work to scan binary?
can generate code and jump to it.
so would have to check thta always running code that you vetted.
check at every indirect jump.

---

eval
what does the fault injection actually do:
uninitialized (i think this just means that load returns random)
bad pointers
null
invert tests

why not just run with real bugs?
- would be good to do to validate
- but not so comprehensive.

How do you evaluate something like this?
Care about whether it improves Reliability, and cost in Performance

How to measure reliability?
Is this realistic?
How do results look?
Too optimistic or pessimistic?

Performance... Let's look at table 4:
why do we have cpu utilization? isn't this captured by overhead?
- no: might have dead spots whre you're waiting and you
can do other things. send-reply.

Play-mp3 looks good
Why does send-stream have more XPCs than receive stream?
Why does this not matter for performance?
Why does compile take bigger hit than send-stream (which has more XPCs)?
Compile is CPU-bound
How did they produce graph in Figure 8?
What is statistical profiling? What does this tell us?
Why don't they show user-mode execution time?
Where is CPU time going?
Extra code -- e.g., XPC, object tracking
Existing code running more slowly?
Why? TLB misses; What are "Pentium 4 performance counters"?
Why does khttpd do so much worse under Nooks? (60% worse, ouch)
15K page/sec -> 6K pages/sec
CPU problem, like compile
Also, transactional, not buffered... how does this affect things?
Do we care? khttpd does sound like a bogus project
Maybe use exokernel/cheetah on dedicated hardware if you care so much..
.

what is difference between 6 and 7?
- why sb so diff? [doesn't do reads, right? so mostly in response
to app]

if it restarts, is system in good state?
[not for vfat: 90% screwed it up. fix? call sync = 10%]

in 7: why not bars to go zero?
eth: unable to send/rec, cannot detect.

non-functioning, but not exception.

why hkttpd has so many crashes?
does a lot at interrupt level (reply to msg there?)
so kill.

what is the cost of an xpc?

what is cost of TLB?

Would same ideas apply to other OSes?
Authors claim Linux is worst case scenario? Why? Do we believe this?
In terms of lots of ill-defined extension interfaces, probably true
That linux doesn't reboot on process-context panic might help, though
Could Nooks be applied to the JOS kernel?