Zero copy “programming” (execution)

Or: How to save memory without (necessarily) influencing programmers.

Stephen Kell

Stephen.Kell@cl.cam.ac.uk

Computer Laboratory

University of Cambridge

Memory is scarce. . .

```
top - 22:00:36 up 8 days,  9:52, 19 users,  load average: 0.37, 0.22, 0.19
Tasks: 197 total,   1 running, 191 sleeping,   1 stopped,   4 zombie
Cpu(s):  6.3%us,  2.0%sy,  0.0%ni, 91.7%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:    513988k total,   507340k used,     6648k free,    25144k buffers
Swap:   999556k total,   359128k used,   640428k free,   138360k cached
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
19048	stephen	20	0	173m	54m	13m	S	0.0	10.8	23:16.18	claws-mail
4875	stephen	20	0	175m	46m	7144	S	0.0	9.2	10:39.89	chrome
4881	stephen	20	0	155m	45m	8496	S	0.0	9.1	11:46.62	chrome
4879	stephen	20	0	148m	34m	8016	S	0.0	7.0	0:47.29	chrome
20078	stephen	20	0	179m	33m	14m	S	0.0	6.6	15:45.38	chrome
3051	root	20	0	315m	27m	3120	S	0.7	5.4	129:39.18	Xorg
19620	stephen	20	0	61280	23m	14m	S	3.6	4.6	1:13.84	vlc
4877	stephen	20	0	140m	17m	6920	S	0.0	3.6	2:54.61	chrome
4884	stephen	20	0	116m	11m	5392	S	0.0	2.4	0:31.16	chrome
20647	stephen	20	0	10368	5604	3176	S	0.0	1.1	0:01.10	nedit
20695	stephen	20	0	6824	4060	1440	S	0.0	0.8	0:00.63	bash
13339	stephen	20	0	6840	4008	1416	S	0.0	0.8	0:00.77	bash
13338	stephen	20	0	8632	3732	1992	S	0.0	0.7	0:00.33	xterm
20694	stephen	20	0	8632	3508	2396	S	0.0	0.7	0:00.16	xterm
18937	stephen	20	0	18892	3108	1568	S	0.3	0.6	0:38.08	fvwm
18933	stephen	20	0	31888	2784	1924	S	0.0	0.5	1:56.73	kded
19044	stephen	20	0	9032	2564	1052	S	0.3	0.5	0:02.20	xterm

Varnish versus Squid

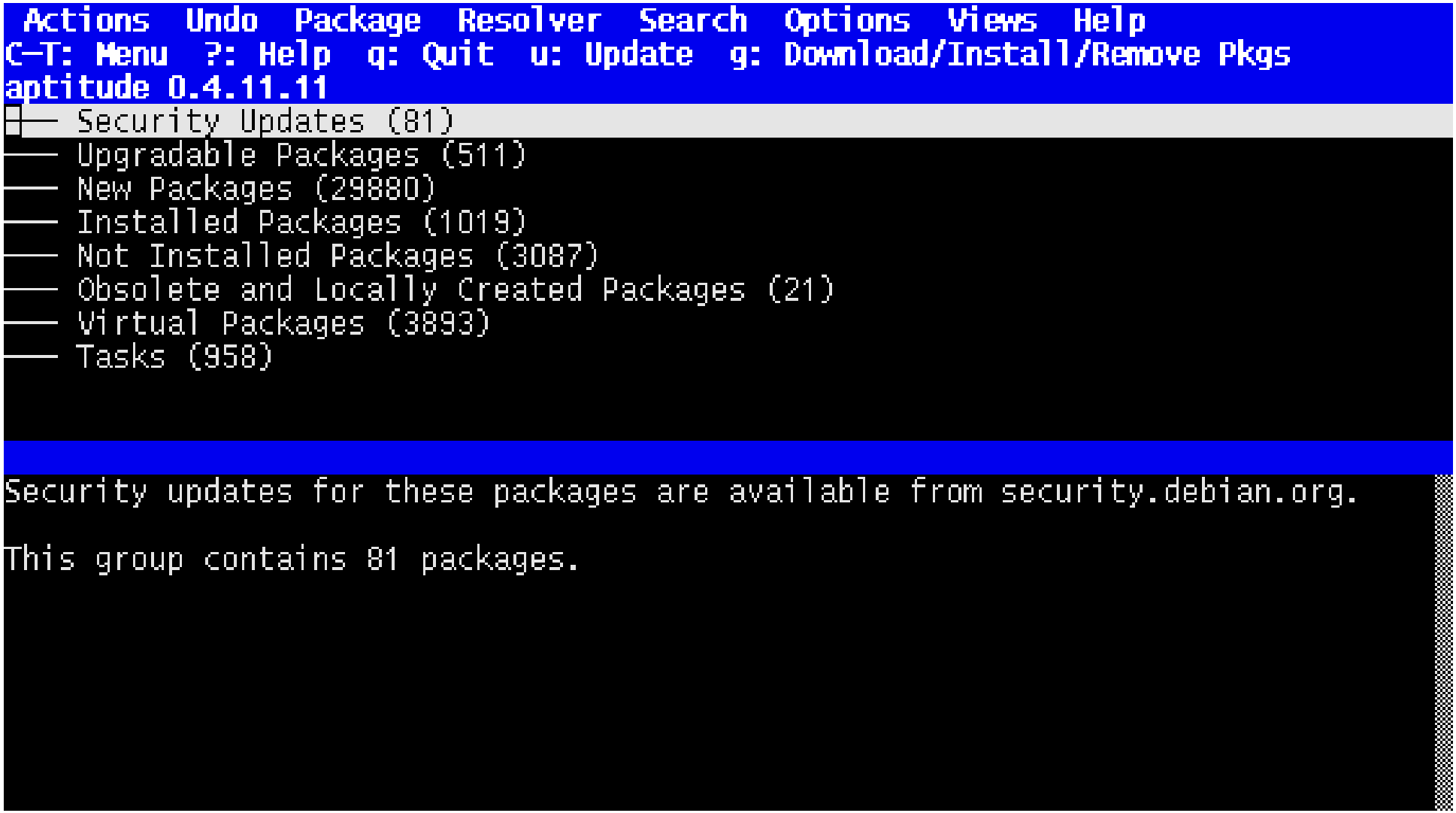
“Computers do not have two kinds of storage any more.”

(Poul-Henning Kamp)

- Don’t explicitly move data between disk and memory.
- Intention: eliminate inefficient app–VM-subsys interaction
- Bonus: smaller virtual memory footprint.

Caveat: data must be identical on disk and memory (*verbatim copy*).

Can we generalise?



Uses a ton of memory storing on-disk data… but not *verbatim*.

The plan

We want to

- avoid allocating memory…
- … if the contents can be reconstructed…
- … from other memory or memory-mappable storage.

Levels of attack:

- programmer behaviour
- programming language definitions
- programming language implementations
- **OS + runtime memory management**

First pass

- “virtualise” culprit malloc() sites
- …no phys. mem is allocated, just faulting VAS region
- on trap, use a *lens* on some *backing* data
- program-demanded representation is still stored…
- but only temporarily; made *maximally ephemeral*

Problems:

- characterise “reconstructed” (any computation?)
- inferring lens logic (+ view update problem)
- locality and page granularity
- backing goes away
- automatic [dynamic] analysis

Outline dynamic analysis

- identify backing data by trapping writes? But how?
- maybe: say *all* new heap alloc is a “virtual region”
  - we may decide to downgrade it to a “physical region” later
- virtual regions: “copy on *unrepeatable* write”
  - means guessing about the future (the usual OS trick)
- backing regions: “copy on *destructive* write”?
  - If we guessed wrong above, may have to fork a copy :-(
- Focus on malloc-sites causing high %age of usage

Identifying lenses

This is the really hard part.

- Identify which functions write a virtual region
- Trace them? Or just analyse instructions...
- ...to check for purity
- Cache these results (and re-evaluate?)...

Simple example: `sscanf()`

- on disk: `"<key>=<value>;\n"`
- in memory: `struct { int key; float value; };`
- read lens: `sscanf(buf, "%d=%f\n", &s.key, &s.value);`
- Know that `sscanf()` is pure, and `buf` available

Conclusions (what?)

Some questions I didn't get time to answer.

- How much saving are we talking?
- Safety: I *think* purity is the key to safety…
- What patterns of copying behaviour are common?
- Is page granularity a problem?
- Bidirectionality…

Bonus: other applications

Avoiding copy-to-memory enables coherent sharing of persistent objects.