

Virtual Memory Primitives for User Programs

Andrew W. Appel and Kai Li
© 1991

Presented by: Nish Aravamudan
May 17, 2006

Overview

- Background
- Necessary Primitives
- Performance Considerations
- System Design
- Examples

Background

- Virtual memory “traditionally” increases size of process' address space
- Can also perform “tricks” via protection
 - Including userspace handlers for protection violations
 - If certain primitives are available to userspace ...

Necessary Primitives

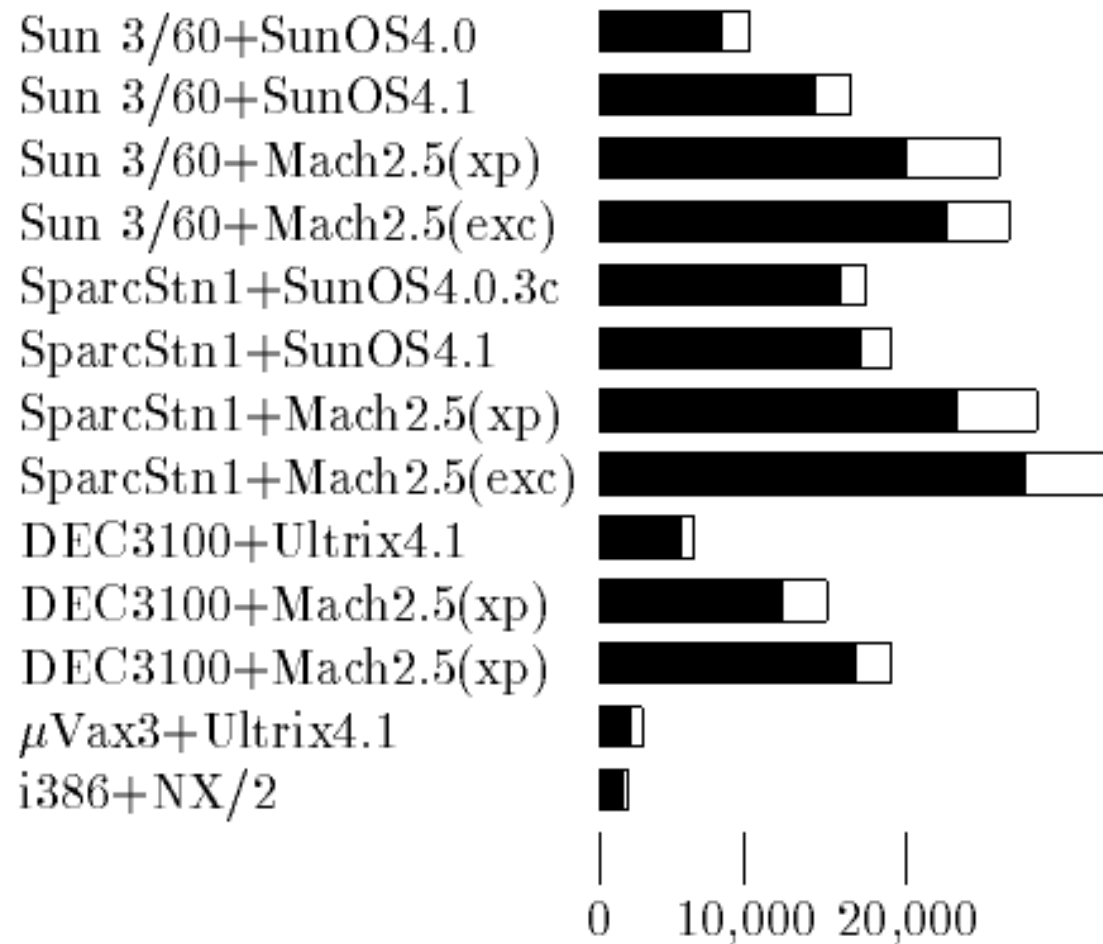
- TRAP – Userspace page-fault handler
- PROT1/PROTN - decrease accessibility of page(s)
- UNPROT – increase accessibility of page
- DIRTY – list of dirtied pages since last call
- MAP2 – map same physical page to two different virtual addresses, with different protections, in same address space

Performance Comparison

- The better the primitives perform, the better the algorithms perform

Machine	OS	ADD	TRAP	TRAP +PROT1 +UNPROT	TRAP +PROTN +UNPROT	MAP2	PAGESIZE
Sun 3/60	SunOS 4.0	0.12	760	1238	1016	yes	8192
Sun 3/60	SunOS 4.1	0.12		2080	1800	yes	8192
Sun 3/60	Mach 2.5(xp)	0.12		3300	2540	yes	8192
Sun 3/60	Mach 2.5(exc)	0.12		3380	2880	yes	8192
SparcStn 1	SunOS 4.0.3c	0.05		*919	*839	yes	4096
SparcStn 1	SunOS 4.1	0.05	†230	1008	909	yes	4096
SparcStn 1	Mach 2.5(xp)	0.05		1550	1230	yes	4096
SparcStn 1	Mach 2.5(exc)	0.05		1770	1470	yes	4096
DEC 3100	Ultrix 4.1	0.062	210	393	344	no	4096
DEC 3100	Mach 2.5 (xp)	0.062		937	766	no	4096
DEC 3100	Mach 2.5 (exc)	0.062		1203	1063	no	4096
μ Vax 3	Ultrix 2.3	0.21	314	612	486	no	1024
i386 on iPSC/2	NX/2	0.15	172	302	252	yes	4096

Graphical Performance Comparison



Examples' Primitive Usage

Methods	TRAP	PROT1	PROTN	UNPROT	MAP2	DIRTY	PAGESIZE
Concurrent GC	✓		✓	✓	✓		✓
SVM	✓	✓		✓	✓		✓
Concurrent checkpoint	✓		✓	✓		‡	✓
Generational GC	✓		✓	✓		‡	✓
Persistent store	✓	✓		✓	✓		
Extending addressability	✓	*	*	✓	✓		✓
Data-compression paging	✓	*	*	✓	✓		
Heap overflow	✓		†				

Concurrent Garbage Collection

- Synchronize collector and mutator
- Based on Baker's algorithm
 - At beginning of collection, all objects in from-space, to-space is empty
 - Collector traces graph of reachable objects, copying each into to-space
 - Every time mutator allocates new object, invoke collector

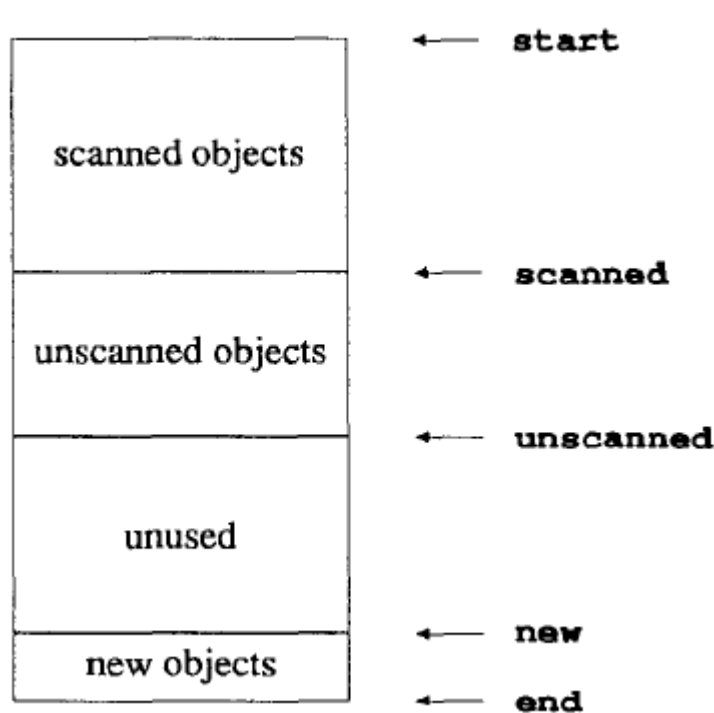
Concurrent Garbage Collection

- Baker's algorithm's invariants
 - Mutator only sees to-space pointers in registers
 - Objects in new and scanned areas only contain to-space pointers
 - Objects in unscanned area contains both to- and from-space pointers

Concurrent Garbage Collection

- VM protections detect from-space references and synchronize mutator/collector
 - Unscanned area set to no-access
 - Mutator will get page-access trap on unscanned objects
 - Collector handles trap, copying objects and forwarding pointers
 - Collector unprotects page and restarts mutator
 - Collector also scans unscanned area concurrently with mutator execution

Concurrent Garbage Collection



To-space
From reference [4]

Collector

Copies from-space objects to unscanned area

Scans objects for pointers to from-space objects

Copies object to unscanned area, if necessary

Forwards pointers to to-space

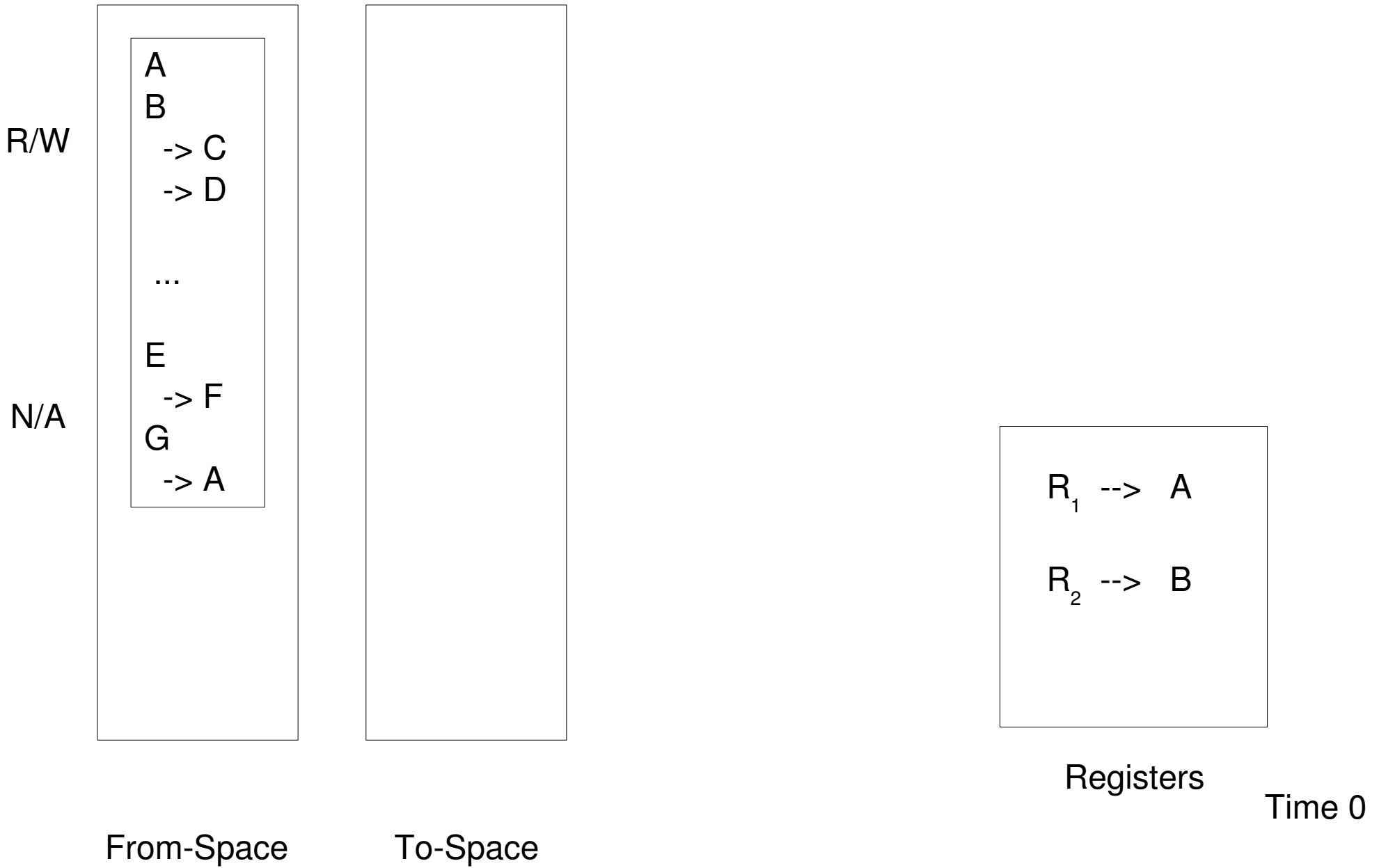
Once scanned == unscanned, all of from-space is garbage

Mutator

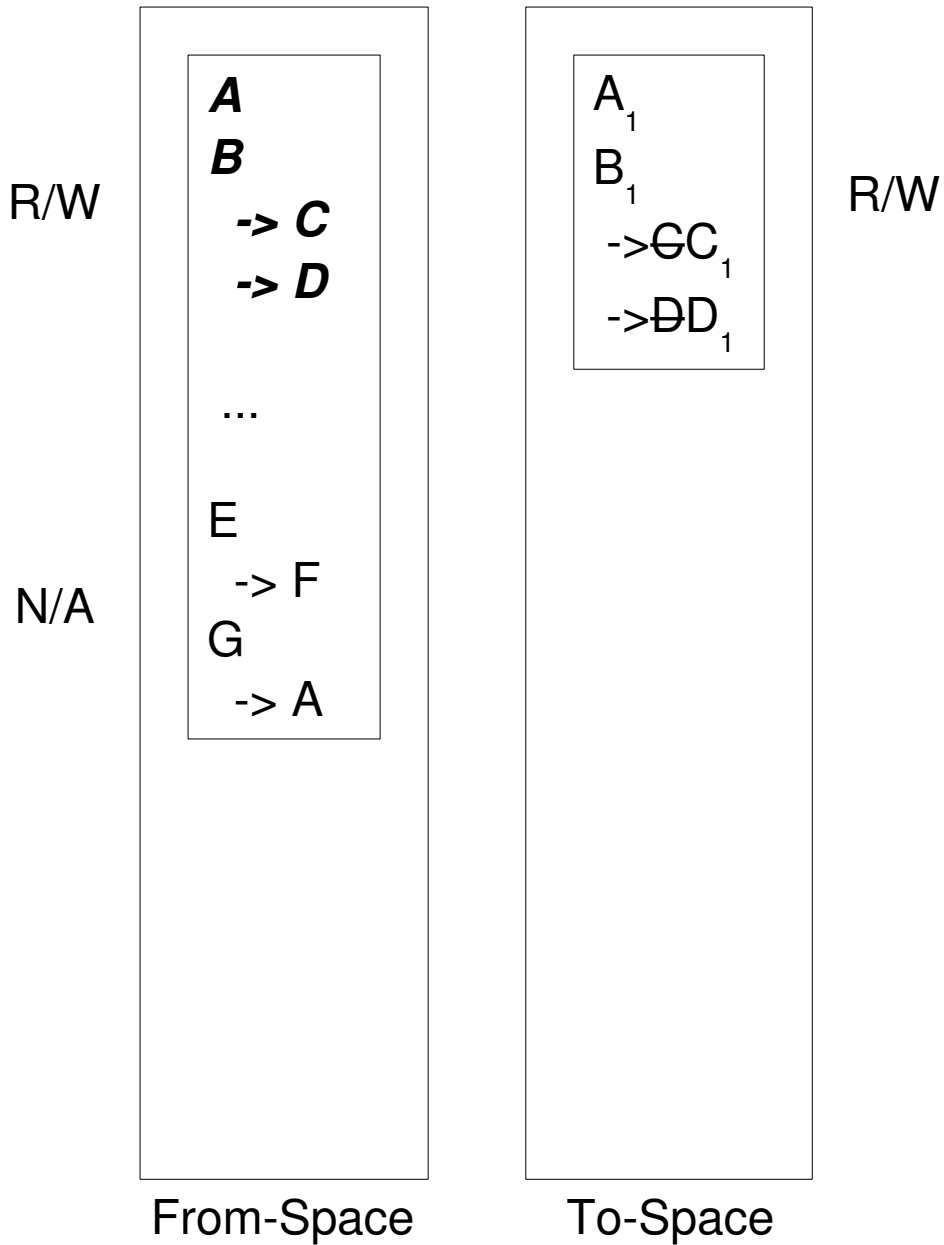
Allocates at new

In Baker's version, blocks while Collector runs

Concurrent Garbage Collection

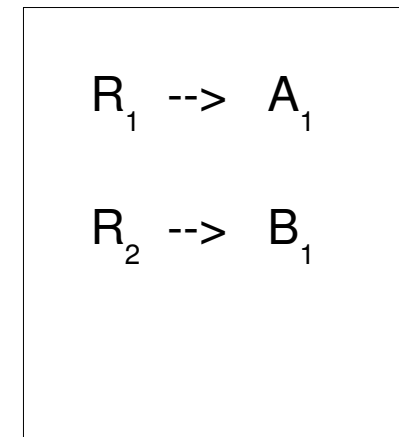


Concurrent Garbage Collection



Collector:

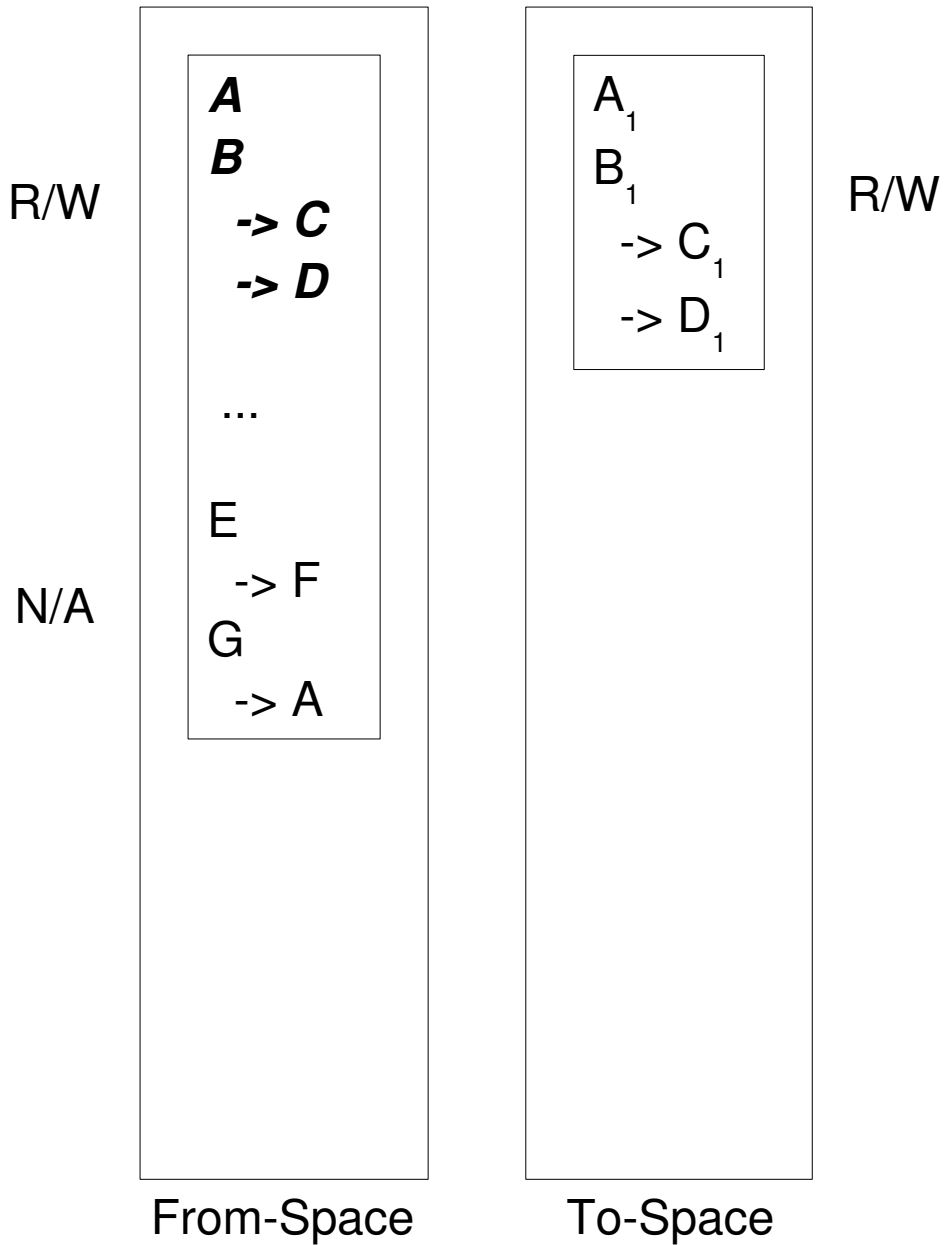
Copies reachable objects (from registers) to to-space
Forwards pointers stored in registers



Registers

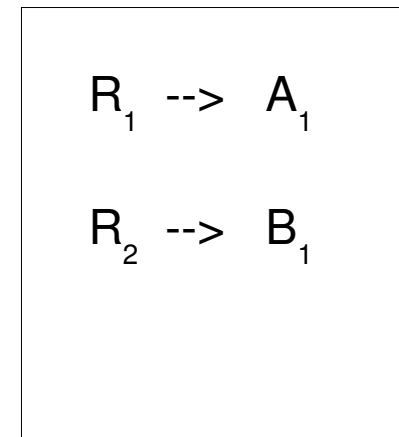
Time 1

Concurrent Garbage Collection



Collector:

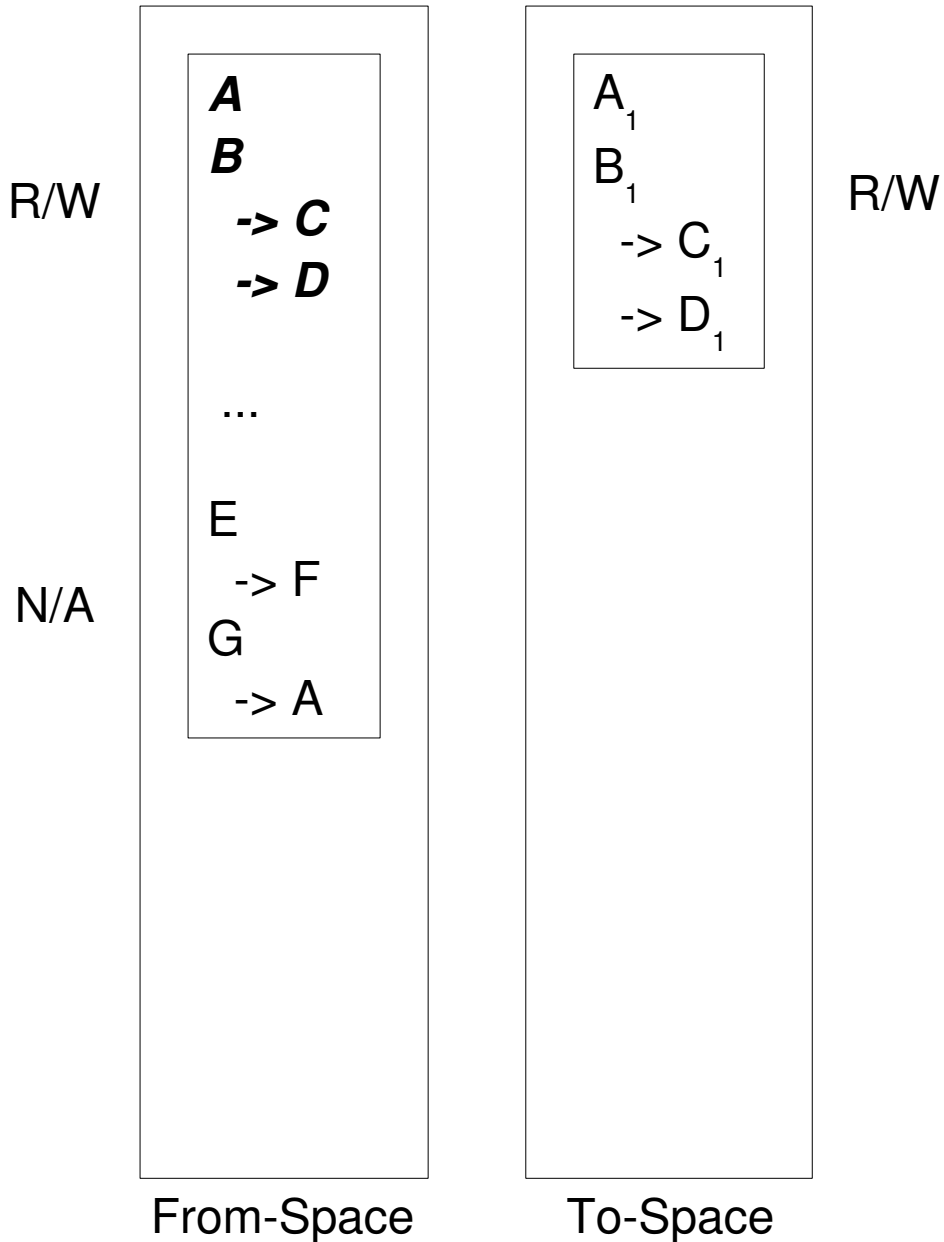
Protects unscanned
area
Starts mutator



Registers

Time 2

Concurrent Garbage Collection

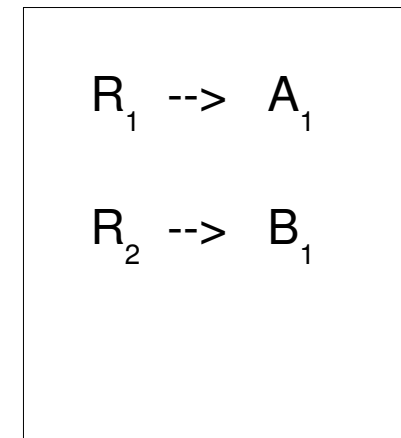


Mutator:

Uses R_1, R_2

Collector:

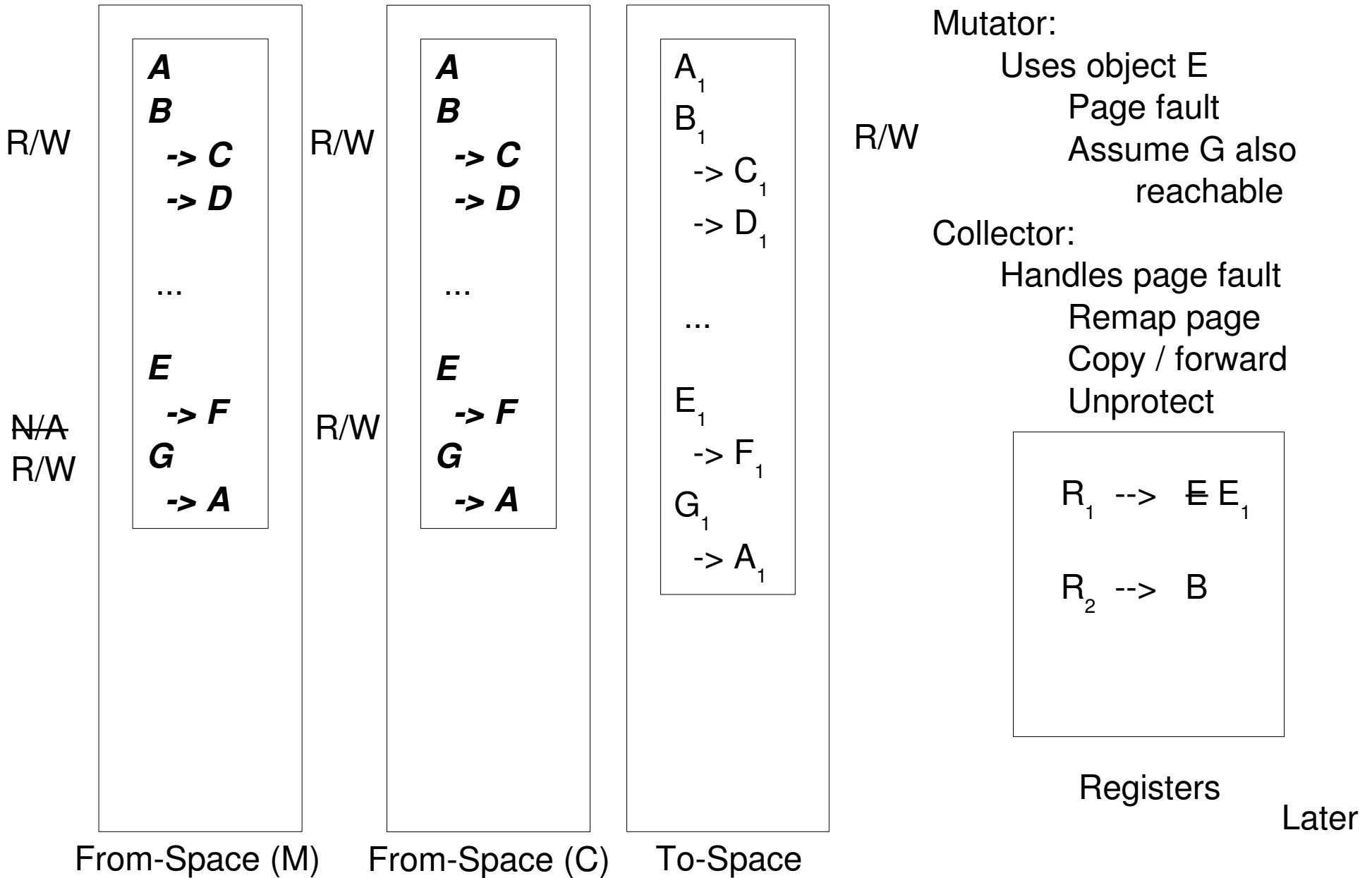
Concurrently scans
next page, copies
reachable objects and
forwards pointers



Registers

Time 3

Concurrent Garbage Collection



Shared Virtual Memory

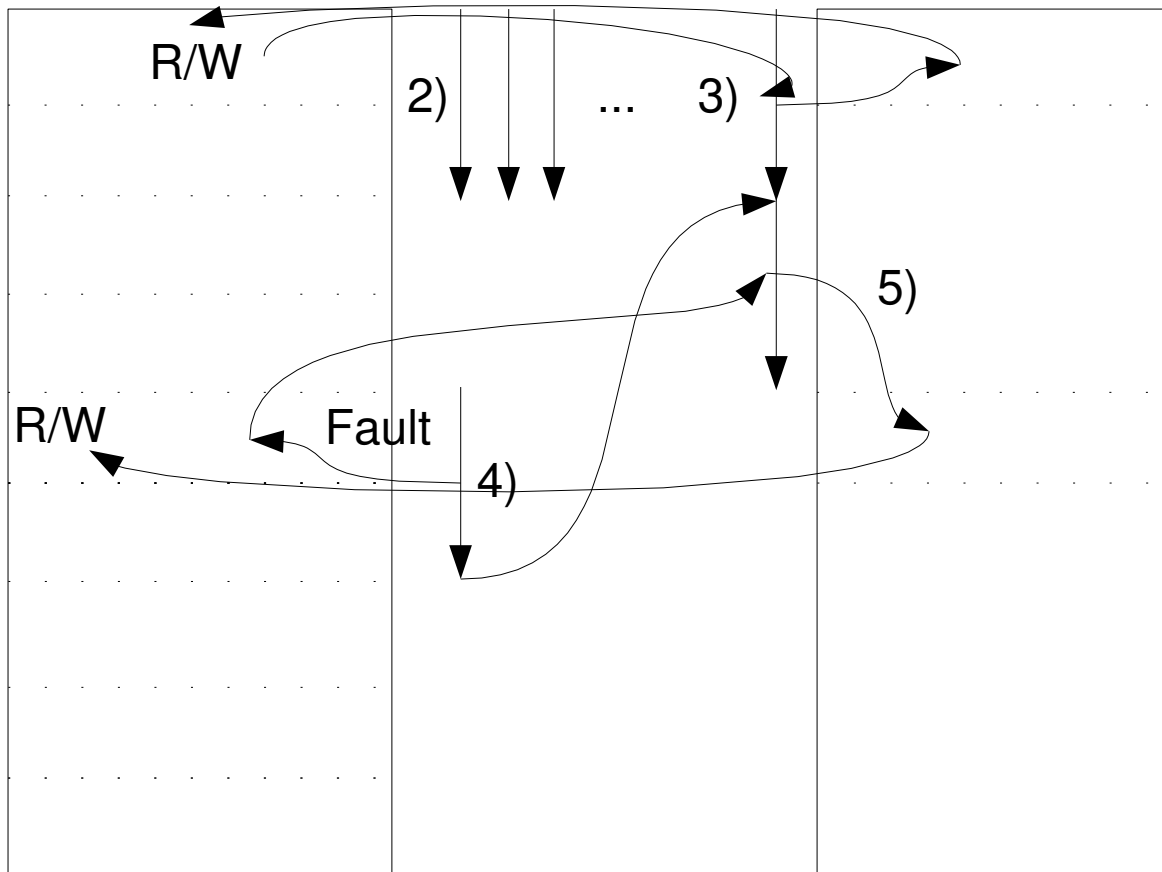
- Multiple processor/memory nodes, connected by fast message-passing network
- SVM presents large coherent shared memory address space
- Maintain single-writer and multiple-reader coherence at page level
 - Many copies of read-only pages, one copy of read-write pages

Concurrent Checkpointing

- Overlap time-consuming checkpoint operations to overlap with program execution
 - 1) Stop all threads in program
 - 2) Heap, globals, stacks saved
 - Set address space to read-only
 - Restart threads
 - Copying thread sequentially scans address space, copying pages to new addresses, then sets access to read-write
 - 3) If restarted thread writes page, gets trap

Concurrent Checkpointing

1) R/W --> R/O



- 1) Mark address space read-only
- 2) Restart application threads
- 3) Copier thread copies pages to separate space
- 4) Thread accesses uncopied page
- 5) Copier handles page fault
- 6) Copier restarts thread

Generational Garbage Collection

- Two properties of dynamically allocated records
 - 1) Younger records more likely to die soon
 - 2) Younger records tend to point to older records
- All records in G_i older than G_{i+1}
- 2nd property means few or no pointers from G_i to G_j , $i < j$
- Collect in youngest generation
- Detect assignments to old objects

Persistent Stores

- Dynamic allocation heap persisting from one invocation to the next
 - On stable storage
- Memory-map persistent store disk file
- Application can modify store, but must commit modifications (dirty pages)

Extending Addressability

- Persistent store can exceed 2^{32} objects
 - In any one run, not likely
- Use 64-bit addresses in disk objects and 32-bit addresses in core objects
- Translation table stores 64-bit pointer -> 32-bit pointer translations on pagein
 - Accessing 32-bit pointer may result in page fault

Data-Compression Paging

- Pages of 32-bit words can be compressed to $\frac{1}{4}$ of page
 - Compress instead of paging out
 - Uncompress instead of paging in
- Can be done in OS or in garbage collector

Heap Overflow Detection

- Must protect against overflow of stack
 - Mark pages above stack no-access
 - Kernel usually handles page faults by allocating physical pages for stack transparently
- Heap overflow in garbage-collected system
 - Ordinarily done with compare and conditional-branch
 - Instead, terminate heap with guard page
 - On fault, collect garbage

Primitive Performance Matters

- Algorithms share several common traits
 - Memory made less-accessible in large batches and more-accessible page by page
 - Almost all fault-handling done by CPU and takes time proportional to page size
 - Every page fault makes page more accessible
 - Frequency of faults inversely related to locality
 - User-mode service routines need access to pages protected from user-mode client routines
 - Service routines don't care about client's CPU state

Relevance to System Design

- TLB Consistency with less-accessible pages
 - Batch shutdown of TLB entries
 - Same for paging out
- Optimal page size
 - Fault-handling by CPU in $\Theta(\text{PAGE_SIZE})$
- Access to protected pages
 - Allow one thread access while preventing others
 - Multiple mapping, kernel copy, shared pages or in-kernel thread

Conclusions

- VM design has significant impact on userspace performance and program design
- If you provide good services, applications will run better
 - Perhaps more important to provide the right services
 - Other primitives as well, e.g. pin pages (mlock)
- Also must ensure correctness
 - e.g., mprotect() not flushing TLB