Operating systems

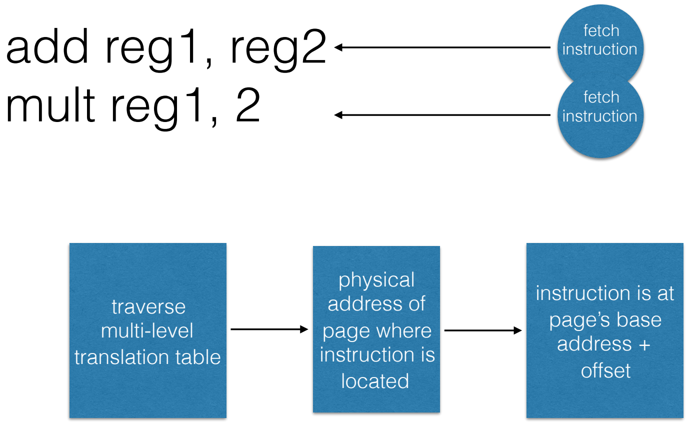
Memory: speeding up address translation
free memory management

Review

What is paging good for?

Today

- Speeding up address translation
- Managing free memory

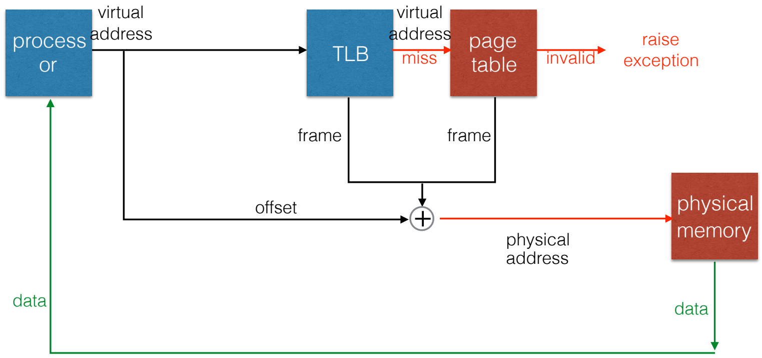


add reg1, reg2

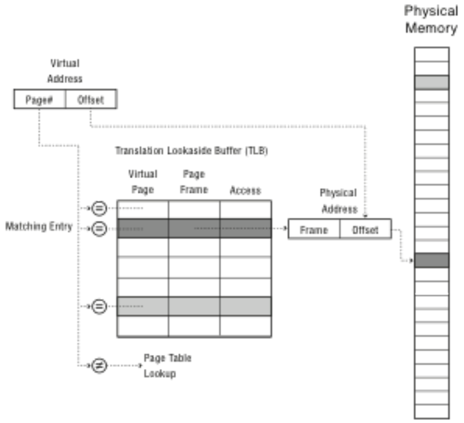
mult reg1, 2

fetch instruction
fetch instruction
traverse multi-level translation table
physical address of page where instruction is located
instruction is at page's base address + offset

Translation lookaside buffer (TLB)

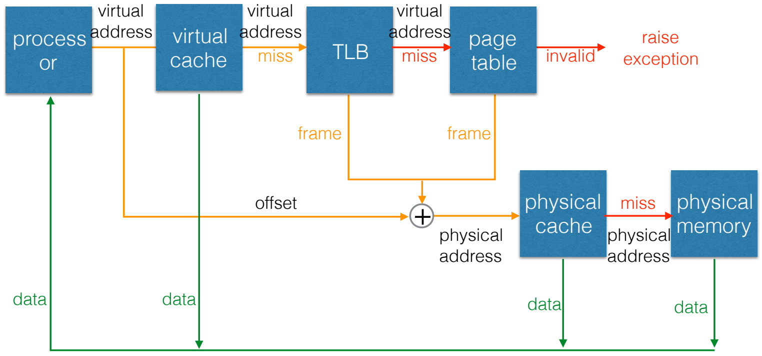


Inside the TLB



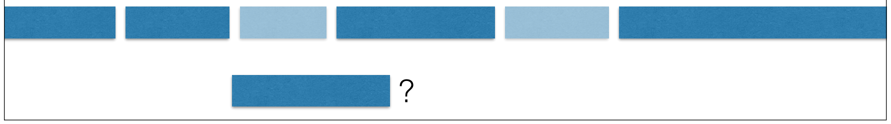
cost of adress translation = cost of TLB lookup + cost of full translation x probability of a miss

Caches galore

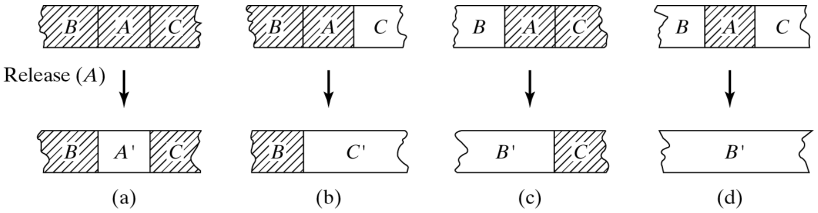


- Caching is a standard technique for speeding up data lookup
- Challenge #1: how to maintain consistency?
- Challenge #2: what to keep in cache, what to kick out?

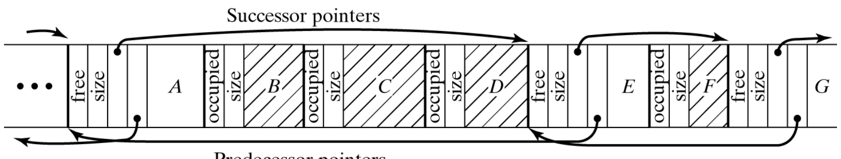
- How do you minimize memory fragmentation?



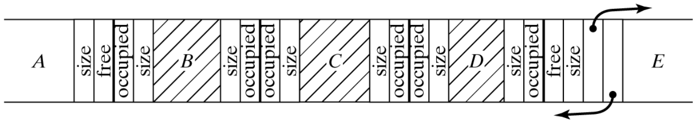
Hole coalescing on a release



Holes as linked lists



(a) with tags at one end



(b) with tags at both ends

Choosing a hole to allocate

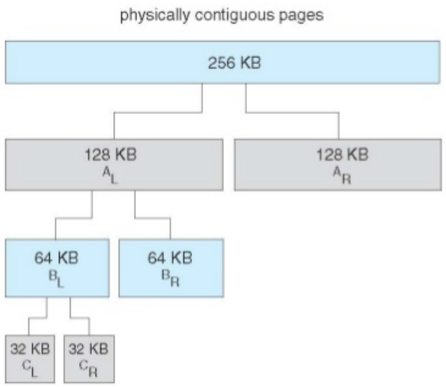
Issues to consider

- utilization
- external fragmentation
- search time

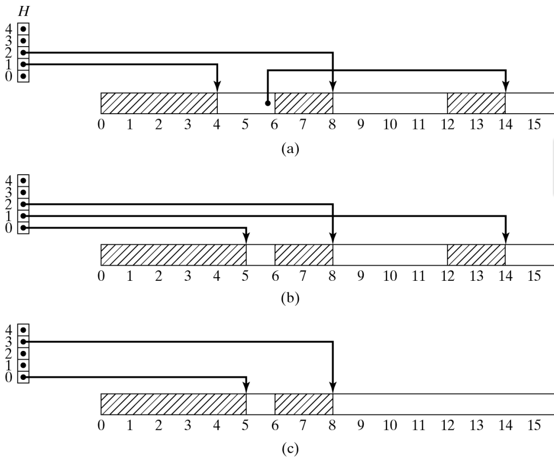
Allocation strategies

- first-fit
- next-fit
- best-fit
- worst-fit

Buddy system



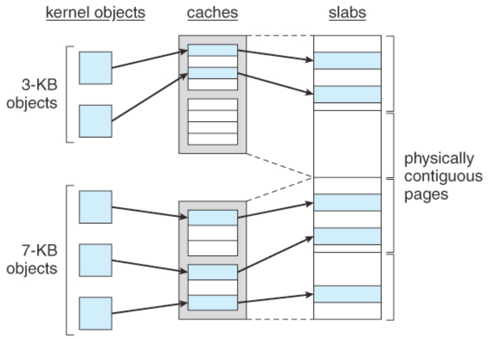
Buddy system: example



1 unit of memory requested

memory at 12 and 13 released

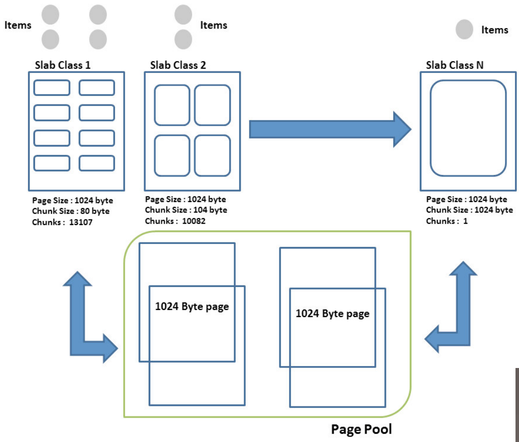
Slab allocation



slab: one or more physically contiguous pages

cache: one or more slabs, one cache per kernel data structure type

Slab allocation

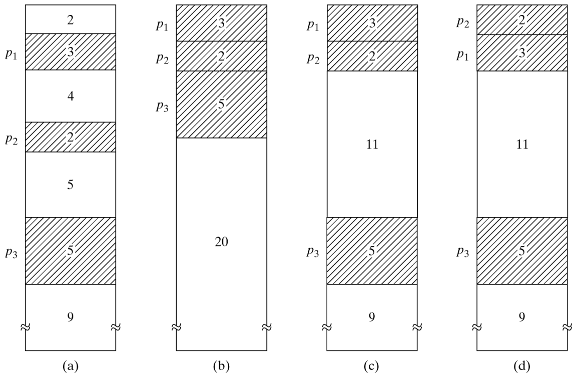


Managing insufficient memory

- Memory compaction
- Swapping
- Overlays

Memory compaction

consolidating smaller holes
dispersed throughout memory
into a single larger hole



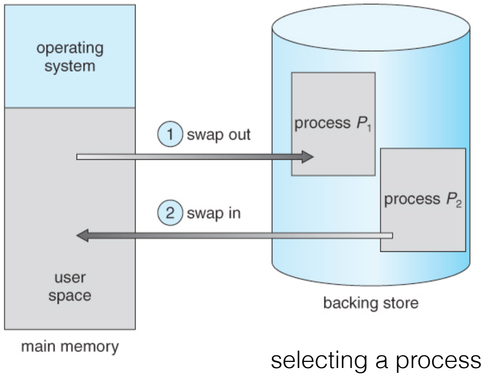
initial state

complete compaction

partial compaction

minimal data movement

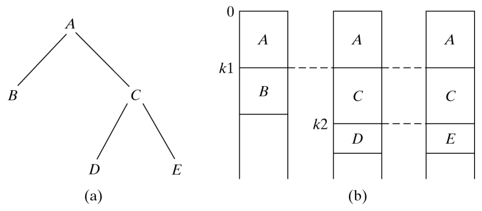
Swapping



selecting a process and
temporarily evicting it
to secondary storage

Overlays

different portions of the program
replace each other in memory
as execution proceeds.



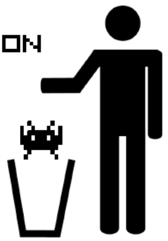
function call
hierarchy

address
assignment

When does free memory need to be managed?

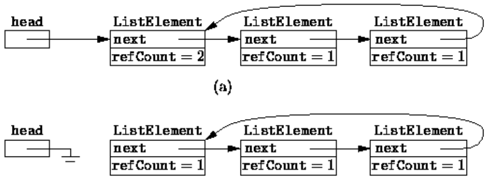
- OS level
  - (early on) in segmented memory systems
  - to manage kernel memory
- program level
  - in languages such as C and C++: malloc/free, new/delete to manage the heap
  - Java has **new** but no **delete**, why?
  - when you know the usage pattern of your application and handling it yourself is faster: eg memcached

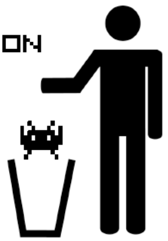
GARBAGE COLLECTION

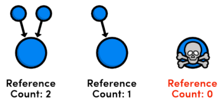


- reference counting
- mark-sweep
- mark-compact
- copying-collector

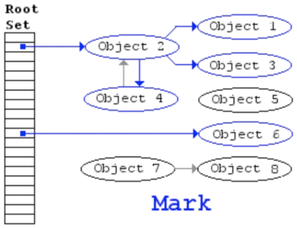
GARBAGE COLLECTION



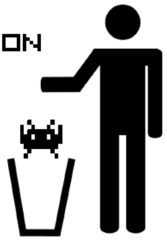


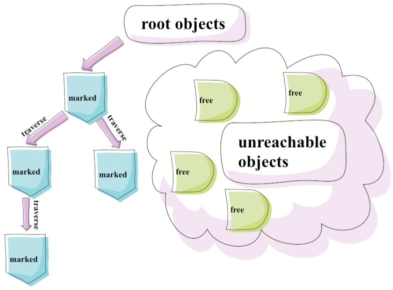


- **reference counting**
- mark-sweep
- mark-compact
- copying-collector



GARBAGE COLLECTION





- reference counting
- **mark-sweep**
- mark-compact
- copying-collector

Recap: memory management

- **efficient and flexible memory use**
  - paging and segmenting system
  - multi-level page tables -> sparse addressing
  - shared code
- **security and isolation**
  - branch and bound
  - read/write access
- **speeding up data retrieval**
  - caching: TLB, virtual and physical caches
- **managing free memory (depends on the actual use)**
  - not a problem if chunks are uniform size
  - techniques: coalescing, hole selection, buddy system, slab allocation
  - language support for managing heap: malloc/free, garbage collection