Λειτουργικά Συστήματα (HY321)

Διάλεξη 10:

Διαχείριση Μνήμης σε Επίπεδο Χρήστη

Χρήστος Δ. Αντωνόπουλος
27/4/2017

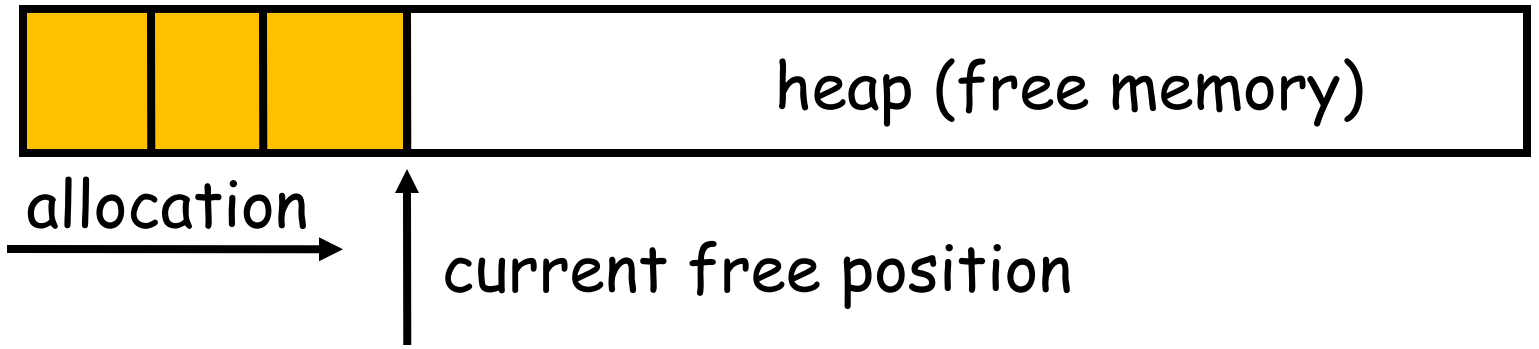
Τμήμα Ηλεκτρολόγων Μηχανικών & Μηχανικών Υπολογιστών, Πανεπιστήμιο Θεσσαλίας

Δυναμική Διαχείριση Μνήμης

- Σχεδόν κάθε πρόγραμμα χρησιμοποιεί δυναμική διαχείριση μνήμης
  - Σημαντική βελτίωση λειτουργικότητας
    - Δεν χρειάζεται να οριστούν στατικά πολύπλοκες δομές δεδομένων
    - Τα δεδομένα μπορούν να μεγαλώνουν συναρτήσει του μεγέθους της εισόδου
  - Μπορεί όμως να γίνει κόλαση για την επίδοση!!!
- Ενδιαφέρουσες παρατηρήσεις:
  - 2-3 αλλαγές στον κώδικα μπορούν να έχουν τεράστια (μη προφανή) επίδραση στην επίδοση του διαχειριστή μνήμης
  - Αδύνατο να έχουμε πάντα τον “τέλειο” διαχειριστή μνήμης
  - Μετά από 35 χρόνια ακόμα δεν καταλαβαίνουμε καλά τη διαχείριση μνήμης

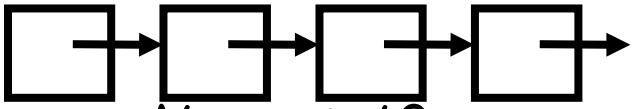
Ποιος είναι ο στόχος; Ποια η δυσκολία;

- Ικανοποίηση “τυχαίας” σειράς δεσμέυσεων / αποδεσμέυσεων
- Εύκολο χωρίς free: Κράτα ένα δείκτη προς μια μεγάλη ελεύθερη περιοχή μνήμης (“heap”) και προχώρησέ τον μετά από κάθε δέσμευση:



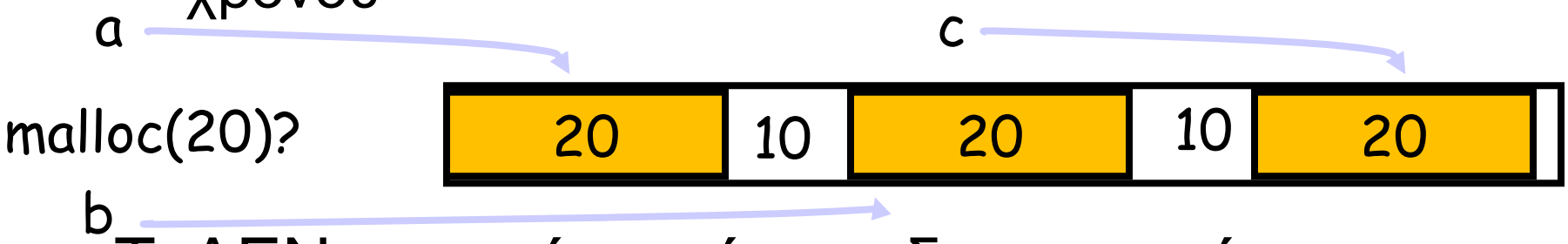
- Πρόβλημα: Το free δημιουργεί τρύπες (“κατακερματισμός”) Αποτέλεσμα? Ελεύθερος χώρος, αλλά οι δεσμεύσεις δε μπορούν να ικανοποιηθούν!

Πιο αφηρημένα



Λίστα ελεύθερων περιοχών

- Τι πρέπει να κάνει ο διαχειριστής;
  - Παρακολουθεί τα ελεύθερα και τα δεσμευμένα τμήματα μνήμης
  - Ιδανικά: χωρίς απώλεια χώρου, χωρίς απώλεια χρόνου



- Τι ΔΕΝ μπορεί να κάνει ο διαχειριστής:
  - Να ελέγξει τη σειρά και το μέγεθος των αιτήσεων
  - Να αλλάξει ptrs του χρήστη => μόνιμες (κακές) αποφάσεις τοποθέτησης.

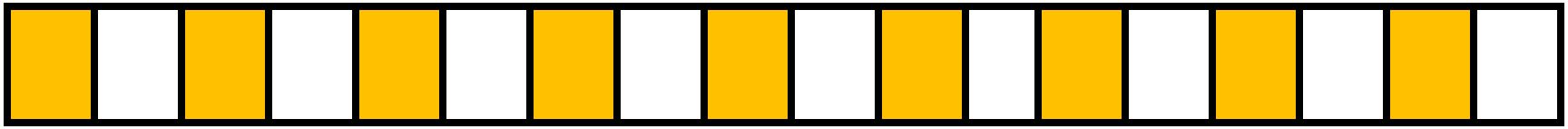
Τι είναι τελικά ο κατακερματισμός;

"Αδυναμία χρήσης ελεύθερης μνήμης"

- 2 αιτίες
  - Διαφορετική διάρκεια ζωής: αν γειτονικά αντικείμενα "πεθάνουν" σε διαφορετικές στιγμές, τότε κατακερματισμός:

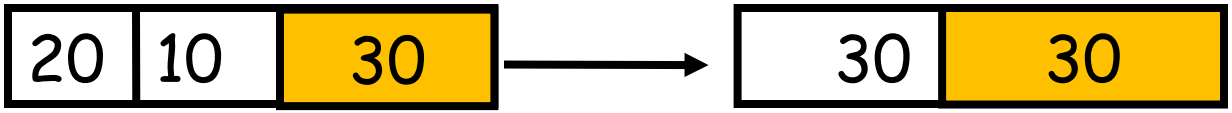
  - Αν πεθάνουν την ίδια στιγμή, εξαφανίζεται ο κατακερματισμός:

  - Διαφορετικά μεγέθη: Αν όλες οι αιτήσεις ήταν του ίδιου μεγέθους, δε θα είχαμε κατακερματισμό (σελιδοποίηση).



Σημαντικές αποφάσεις

- Επιλογή τοποθέτησης: σε ποιο σημείο της ελεύθερης μνήμης να ικανοποιήσω μια αίτηση;
  - Ελευθερία: μπορώ να διαλέξω οποιοδήποτε σημείο
  - Ιδανικά: Βάλτη κάπου που δε θα προκαλέσει κατακερματισμό αργότερα (τη γυάλινη σφαίρα...)
- Σπάσιμο ελεύθερων τμημάτων για να ικανοποιηθούν μικρές αιτήσεις
  - Αντιμετωπίζει τον εσωτερικό κατακερματισμό.
  - Ελευθερία: Μπορώ να σπάσω οποιοδήποτε μεγάλο τμήμα.
  - Ένας τρόπος: διάλεξε το τμήμα που θα δώσει το μικρότερο “υπόλοιπο” (best fit).
- Συνένωση ελεύθερων τμημάτων – δημιουργία μεγαλύτερων



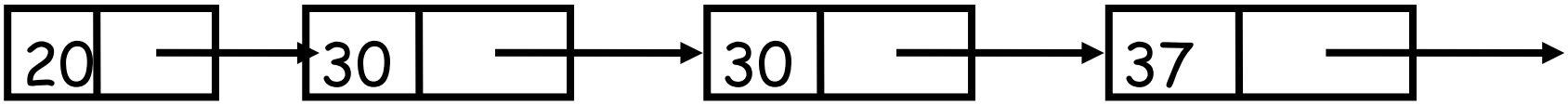
  - Ελευθερία: Πότε θα γίνει η συνένωση (καλό είναι να την καθυστερήσεις)
  - Αντιμετωπίζεται ο εσωτερικός κατακερματισμός

Αδύνατο να εξαφανίσεις τον κατακερματισμό

- Εργασίες για διαχείριση μνήμης:
  - Παζάρια, ζύγισμα θετικών & αρνητικών κάθε λύσης
  - Ο λόγος; Δεν υπαχει ο τέλειος διαχειριστής μνήμης.
- Λίγη θεωρία:
  - Για κάθε αλγόριθμο διαχείρισης, υπάρχει ακολουθία δεσμέυσεων / αποδεσμέυσεων που τινάζουν το διαχειριστή στον αέρα και προκαλούν μεγάλο κατακερματισμό.
  - Τι είναι καλό (αποδεκτό) / τι είναι κακό?
    - Καλός διαχειριστής: M*log(n) όπου M = bytes “ζωντανών” δεδομένων και n = λόγος μεταξύ μικρότερων & μεγαλύτερων μεγεθών
    - Κακός διαχειριστής: M*n

Best fit

- Ιδέα: Ελαχιστοποίησε τον κατακερματισμό δεσμεύοντας μνήμη από το τμήμα που θα δημιουργήσει το μικρότερο υπόλοιπο
- Υλοποίηση: Η heap είναι λίστα ελεύθερων τμημάτων, με μέγεθος / αρχή του καθενός και δείκτη προς το επόμενο.

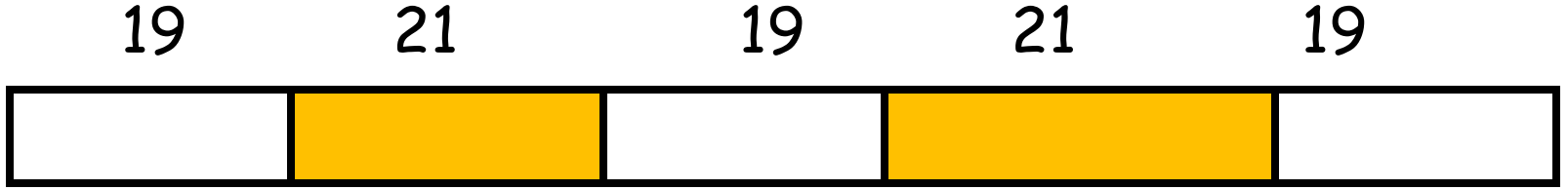


- Κώδικας: Ψάξε τη λίστα για το τμήμα με μέγεθος κοντύτερα στο μέγεθος της αίτησης.
- Στο free (συνήθως) σύμπτυξε γειτονικά blocks
- Πρόβλημα: “Πριονίδι”
  - Το υπόλοιπο τόσο μικρό που με το χρόνο γεμίζει ο τόπος “πριονίδι”.
  - Ευτυχώς στην πράξη δεν είναι και μεγάλο πρόβλημα

Best fit: Η αποκαθήλωση

- Απλή, κακή περίπτωση: Δέσμευσε εναλλάξ n, m (m<n), ελευθέρωσε όλα τα m, και προσπάθησε να δεσμεύσεις ένα m+1.
- Παράδειγμα: Αρχικά 100 bytes ελεύθερης μνήμης
  - alloc 19, 21, 19, 21, 19

  - free 19, 19, 19:



  - alloc 20? Αποτυγχάνει! (χαμένος χώρος = 57 bytes)
- Όμως: Δε φαίνεται να συμβαίνει συχνά στην πράξη (αν και η συμπεριφορά των προγραμμάτων δείχνει ότι θα μπορούσε)

First fit

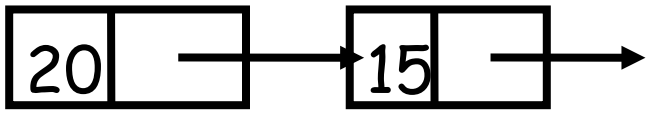
- Ιδέα: διάλεξε το πρώτο ελεύθερο τμήμα στο οποίο χωράει η αίτηση
- Δομή: ελεύθερη λίστα, ταξινομημένη lifo, fifo, ή κατά διεύθυνση
- Κώδικας: Πάρε την 1η περιοχή από τη λίστα
  - LIFO: Βάλε αντικείμενο που απελευθερώνεις στην αρχή της λίστας.
    - Απλό, αλλά προκαλεί μεγαλύτερο κατακερματισμό
  - Ταξινόμηση κατά διεύθυνση:
    - Εύκολες συνεννώσεις (έλεγξε απλά αν το αμέσως επόμενο τμήμα είναι ελεύθερο)
    - Διατηρεί ενιαίο τον ελεύθερο χώρο (καλό)
  - FIFO: Το αντικείμενο που απελευθερώνεις μπαίνει στο τέλος της λίστας.
    - Κατακερματισμός ~= ταξινόμηση κατά διεύθυνση
      - Ποιος ξέρει γιατί...

Παθολογικό παράδειγμα: LIFO FF

- LIFO first fit φαίνεται OK:
  - Ελεύθερο αντικείμενο στην αρχή της λίστας (φθηνό), ας ελπίσουμε ότι θα ξαναχρειαστεί το ίδιο μέγεθος (φθηνό + καλή τοπικότητα).
  - Όμως, προβληματικό για απλά πρότυπα δέσμευσης:
    - Μείγμα μεγάλων δεσμέυσεων με μικρή ζωή με μικρές δεσμεύσεις με μεγάλη ζωή.
    - Κάθε φορά που απελευθερώνεται ένα μεγάλο τμήμα, “κόβεται” σύντομα από αυτό ένα μικρό τμήμα.
      - Παθολογικός κατακερματισμός...

Λεπτομέρειες του First fit

- Στην πράξη First fit με ταξινόμηση διευθύνσεων:
  - Κατά προτίμηση σπάνε τα τμήματα στην αρχή της λίστας. Όσα είναι πιο πίσω σπάνε μόνο αν δε βρεθεί κάποιο μεγαλύτερο μπροστά τους
  - Αποτέλεσμα? Χονδρικά ταξινομείται η λίστα κατά έγεθος
  - Και? Το first fit γίνεται παρόμοιο με το best fit: Το first fit σε ταξινομημένη λίστα = best fit!
  - Πρόβλημα: “πριονίδι” στην αρχή της λίστας
    - Η ταξινόμηση της λίστας αναγκάζει τις μεγάλες αιτήσεις να προσπεράσουν πολλά μικρά τμήματα. Χρειάζεται αποδοτική οργάνωση της heap.
- Σύγκριση με το best fit?
  - Έστω τα διπλανά ελεύθερα τμήματα:
  - Δεσμεύσεις: 10, μετά 20
  - Δεσμεύσεις: 8, 12, και τέλος 12



First και best fit: Βίοι παράλληλοι

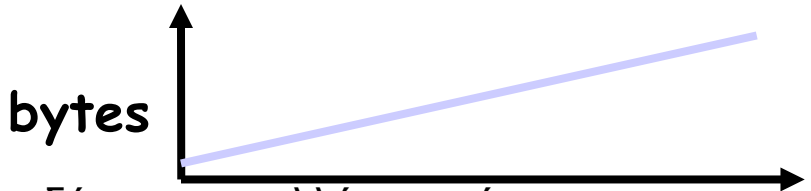
- Φαίνεται να έχουν παρόμοια επίδοση
  - Στην πραγματικότητα παρόμοιες αποφάσεις δέσμευσης τόσο για τυχαία όσο και για πραγματικά υπολογιστικά φορτία!
  - Ποιος να ξέρει γιατί...
  - Μυστήριο γιατί αρχικά φαίνονται αρκετά διαφορετικοί
- Πιθανές εξηγήσεις:
  - First fit ~ best fit γιατί με το χρόνο η ελεύθερη λίστα ταξινομείται κατά μέγεθος: Μικρά αντικείμενα στην αρχή της λίστας, οπότε οι δεσμεύσεις πλησιάζουν το best fit.
  - Και στα δύο εφαρμόζουμε έμμεσα το “open space heuristic”: Μη σπας τα μεγάλα τμήματα.
  - Τα μεγάλα τμήματα χρησιμοποιούνται μόνο όταν είναι απαραίτητο

Κάποιες μάλλον άσχημες ιδέες

- Worst-fit:
  - Στρατηγική: καταπολέμησε το “πριονίδι” χωρίζοντας μεγάλα τμήματα για να έχεις και μεγάλα υπόλοιπα
  - Στο τέλος δε θα έχεις μεγάλα τμήματα... Εγγύηση...
- Next fit:
  - Ιδέα: Χρησιμοποίησε first fit, αλλά θυμήσου που βρήκες το τελευταίο τμήμα που χρησιμοποίησες και ξεκίνα από εκεί.
  - Φαίνεται καλή ιδέα, αλλά τείνει να κατα-σπάσει ολόκληρη τη λίστα.
- Buddy allocation:
  - Στρογγυλοποίησε τις δεσμεύσεις σε δυνάμεις του 2 για να γίνει ευκολότερη & γρηγορότερη η διαχείριση.
  - Αποτέλεσμα: Άσχημος εσωτερικός κατακερματισμός.

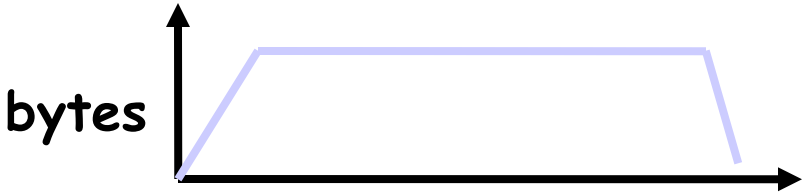
Πρότυπα πραγματικών προγραμμάτων

- Τα προγράμματα δεν είναι ακριβώς “μαύρα κουτιά”.
- Τα περισσότερα συμπεριφέρονται όπως στα παρακάτω σχήματα (ή συνδυασμούς τους):
  - ράμπες: συγκέντρωσε μνήμη μονοτονικά αυξανόμενη με το χρόνο

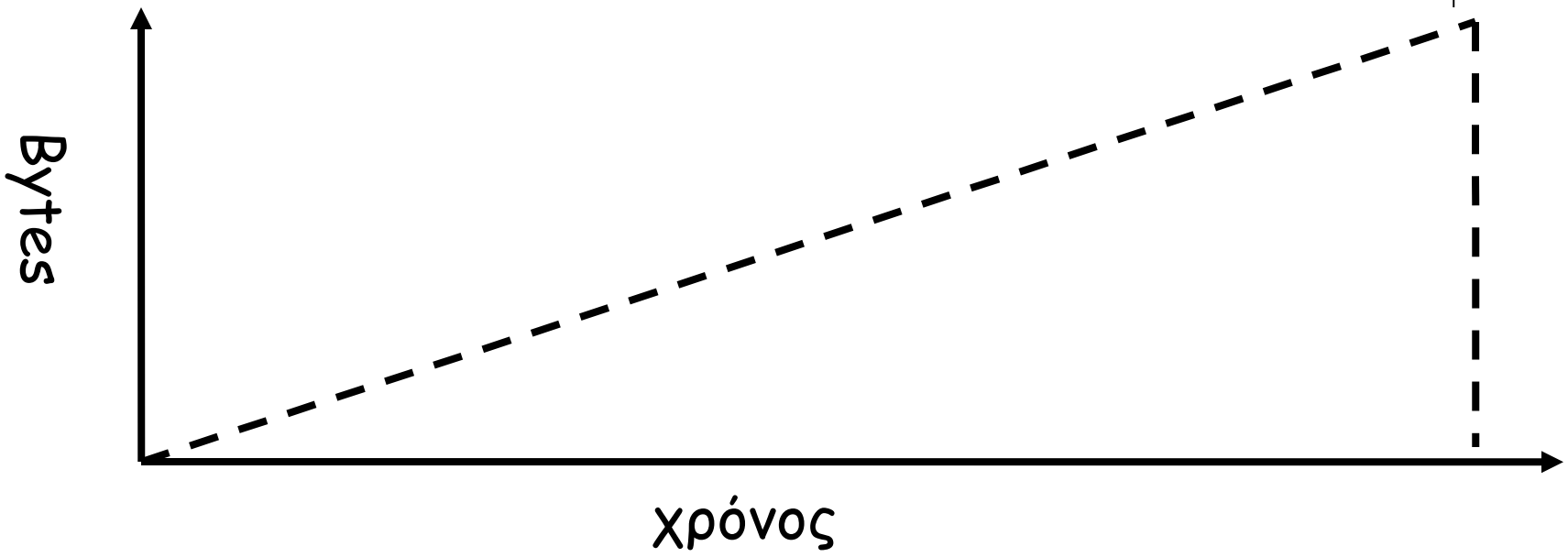


  - κορυφές: δέσμευσε πολλά αντικείμενα, χρησιμοποίησέ τα λίγο, και μετά απελευθέρωσέ τα όλα

  - οροπέδια: δέσμευσε πολλά αντικείμενα, χρησιμοποίησέ τα για ώρα



Ράμπες



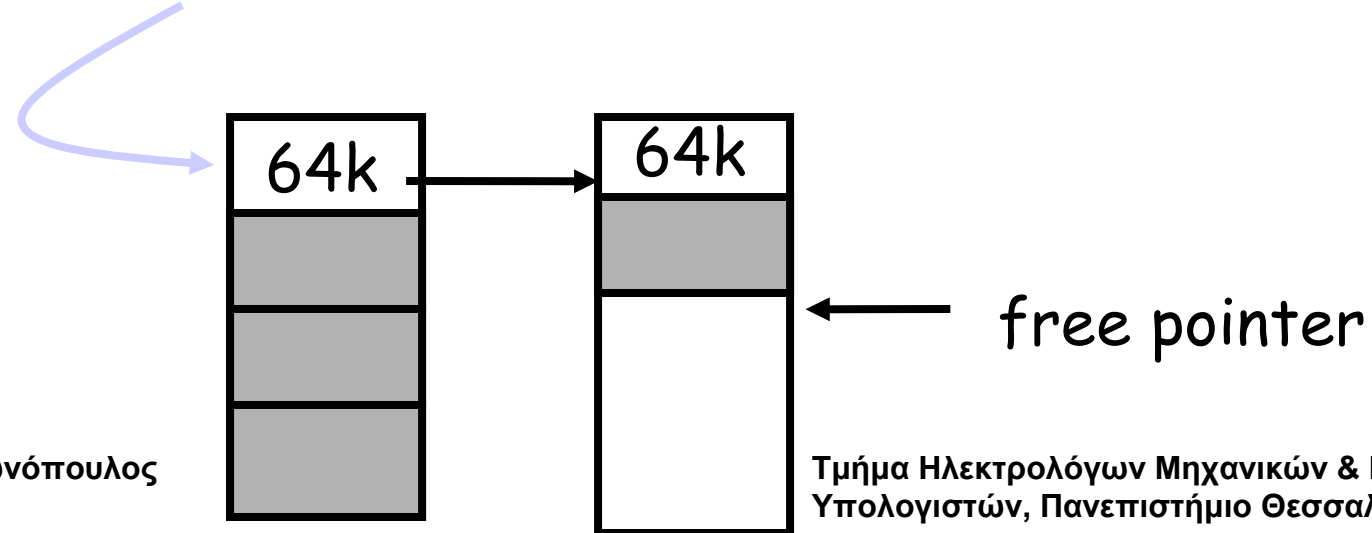
- Στην πράξη: ramp = όχι απελευθερώσεις μνήμης!
- Συνέπειες για τον κατακερματισμό?
- Τι συμβαίνει αν αξιολογούμε διαχειριστές μόνο με προγράμματα με αυτή τη συμπεριφορά ?

Κορυφές

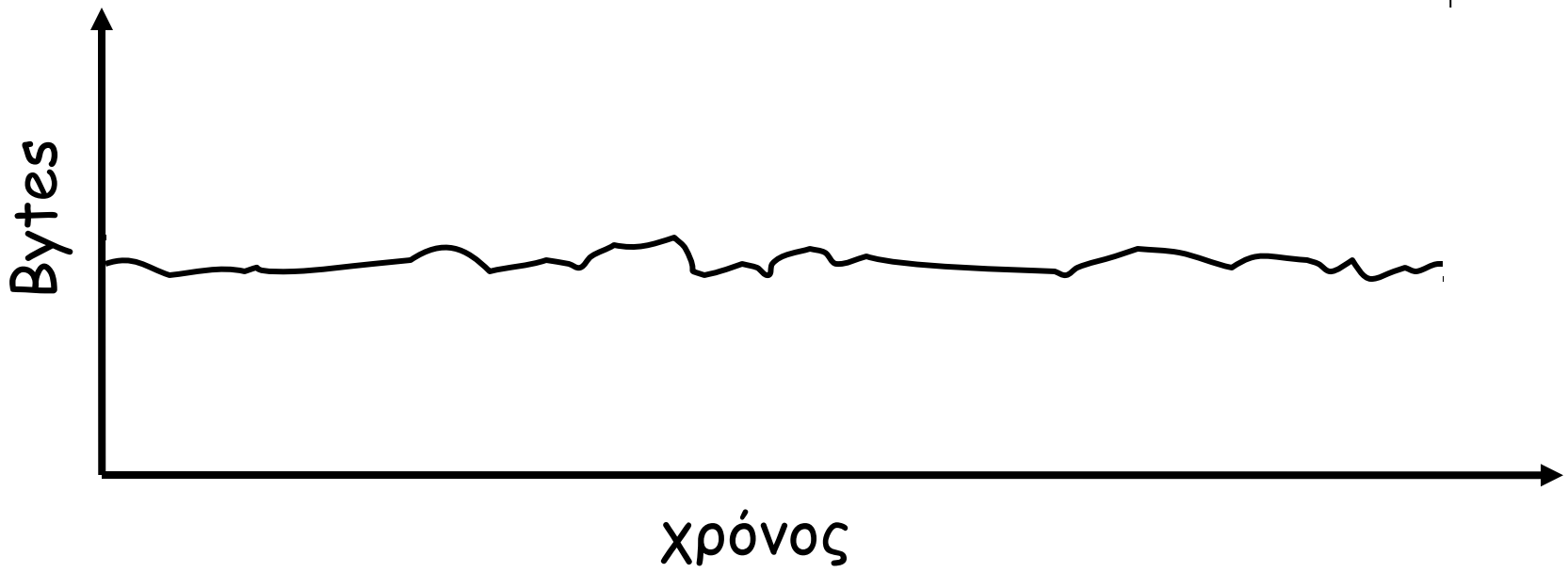
- Κορυφές: Δέσμευσε πολλά αντικείμενα, χρησιμοποίησέ τα για λίγο, απελευθέρωσέ τα όλα
- Μεγάλος κίνδυνος κατακερματισμού.
- Τι θα γινόταν αν ένα πρόγραμμα με συμπεριφορά “κορυφών” δέσμευε από συνεχόμενη μνήμη ?

Αξιοποίηση κορυφών

- Φάσης κορυφής: Πολλές δεσμεύσεις και μετά απελευθέρωση όλων
- Ας φτιάξουμε μια νέα διεπαφή: δεσμεύσεις όπως πριν, αλλά υποστήριξε μόνο ομαδική απελευθέρωση.
  - Λέγεται “arena allocation”, “obstack” (object stack), ή κλήση συνάρτησης (από τους μεταγλωττιστάδες).
  - arena = διασυνδεμένη λίστα μεγάλων περιοχών ελεύθερης μνήμης
  - Πλεονεκτήματα: δέσμευση = προχωράει ένας δείκτης, απλή απελευθέρωση, σχεδόν μηδενική διαχείριση και επιβάρυνση χώρου



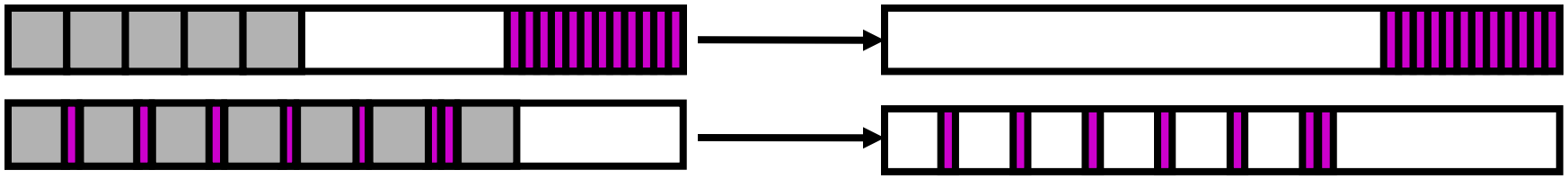
Οροπέδια



- Plateaus: Δέσμευσε πολλά αντικείμενα, χρησιμοποίησε για ώρα

Πολεμώντας τον κατακερματισμό

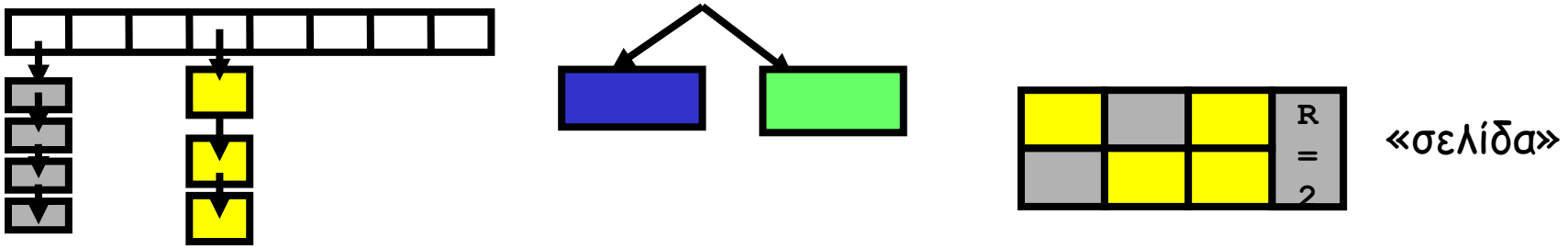
- Ομαδοποίηση = Μείωση κατακερματισμού:
- Δέσμευση μαζί ~ Απελευθέρωση μαζί
- Διαφορετικός τύπος ~ απελευθέρωση ξεχωριστά



- Παρατηρήσεις για την υλοποίηση:
  - Τα προγράμματα δεσμεύουν αντικείμενα από λίγες κλάσεις μεγεθών. Οι περισσότερες δεσμεύσεις για μικρά αντικείμενα (< 10 λέξεις)
  - Ο κατακερματισμός όταν έχουμε μεγάλη χρήση μνήμης πιο σημαντικός από τον κατακερματισμό σε μικρή χρήση μνήμης
  - Οι «εργασίες» στη μνήμη αυξάνουν με το μέγεθός της.
- Συνέπειες;

Απλές, ομαδοποιημένες λίστες ελεύθερων αντικειμένων

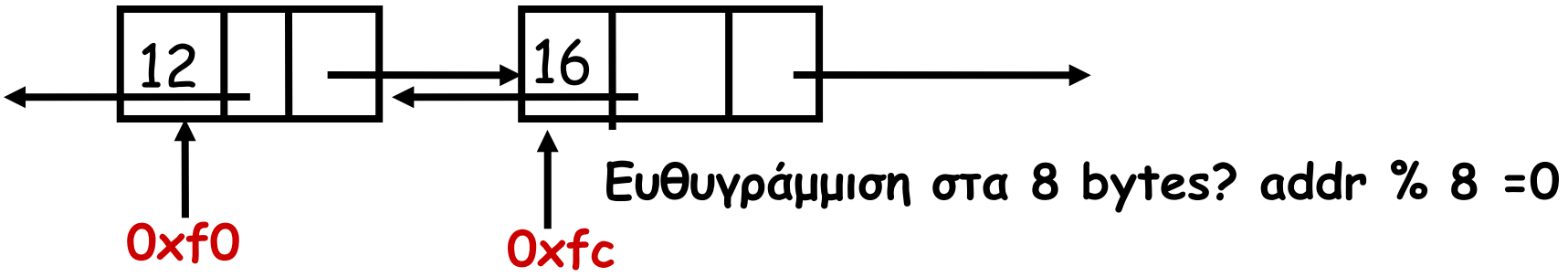
- Πίνακας λιστών ελευθέρων για μικρά μεγέθη, δέντρο για μεγαλύτερα



- Αντικείμενα ίδιου μεγέθους στην ίδια «σελίδα». Μετρητής δεσμευμένων αντικειμένων
  - Αν 0, η σελίδα ελεύθερη
- Πλεονεκτήματα
  - Ομαδοποίηση κατά μέγεθος
  - Όχι μετα-πληροφορία στο αντικείμενο
  - Γρήγορος και μικρός κώδικας δέσμευσης
- Μειονεκτήματα
  - Blowup χειρότερης περίπτωσης: 1 «σελίδα» / μέγεθος

Τυπικές Επιβαρύνσεις Χώρου

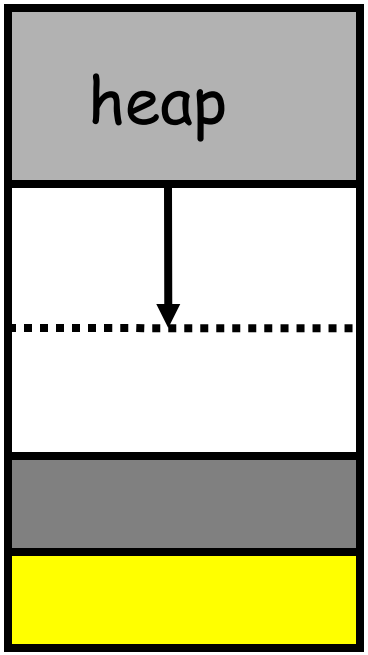
- Επιβαρύνσεις για λογιστικά λίστας + ευθυγράμμιση καθορίζουν το ελάχιστο μέγεθος αντικειμένου:
  - Μέγεθος του block.
  - Δείκτες προς επόμενο & προηγούμενο στοιχείο της ελεύθερης λίστας



- Επιβάρυνση από το σύστημα: ευθυγράμμιση.
  - Ο διαχειριστής δεν ξέρει τον τύπο. Πρέπει να ευθυγραμμίσει «συντηρητικά».
- Ελάχιστη μονάδα δέσμευσης; Επιβάρυνση χώρου κατά τη δέσμευση;

Πώς δίνεται η μνήμη από το Λ.Σ.

- Στο Unix η sbrk αυξάνει τη heap της διεργασίας



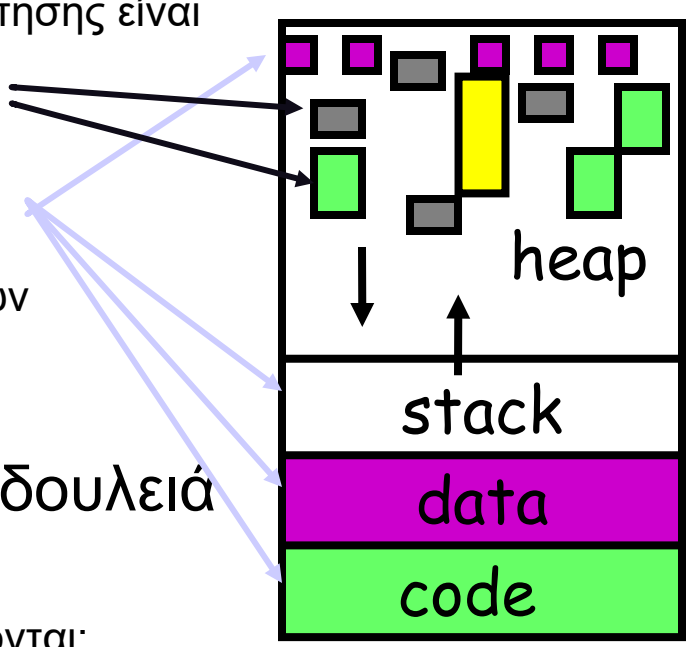
sbrk(4096)

```
/* add nbytes of valid virtual address space */
void *get_free_space(unsigned nbytes) {
  void *p;
  if(!(p = sbrk(nbytes)))
        error("virtual memory exhausted");
  return p;
}
```

- Ενεργοποιεί μια (ή και παραπάνω) «μηδενισμένη» σελίδα του ιδεατού χώρου διευθύνσεων
- Απενεργοποιείται από τον ιδεατό χώρο διευθύνσεων με sbrk(-nbytes)

Διαχείριση μνήμης στο Λ.Σ. και στο επίπεδο χρήστη

- Επανατοποθέτηση (μετακίνηση):
  - Η ιδεατή μνήμη επιτρέπει στο Λ.Σ. να μετακινήσει κομμάτια στη φυσική μνήμη (ανανέωση πίνακα σελίδων, άρα και να συμπιέσει τη μνήμη.
  - Σε επίπεδο χρήστη αδύνατο. Οι αποφάσεις τοποθέτησης είναι μόνιμες
- Μέγεθος και κατανομή:
  - ΛΣ: Λίγα μεγάλα αντικείμενα.
  - Χρήστη: Πολύ μεγάλος αριθμός μικρών αντικειμένων
    - Σημαντικός ο εσωτερικός κατακερματισμός
    - Σημαντική η ταχύτητα δέσμευσης / αποδέσμευσης
- Πολλά αντίγραφα δομών για την ίδια δουλειά
  - Η διαχείριση μνήμης επιπέδου χρήστη χτίζεται πάνω από την ιδεατή μνήμη. Γιατί να μη συνεργάζονται;



Επιστροφή μνήμης: πέρα από το free

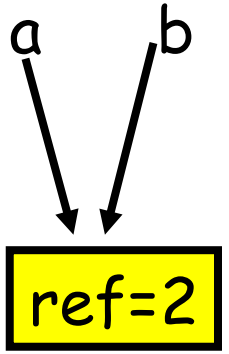
- Αυτόματη επιστροφή:
  - Επίπεδο χρήστη: Ότι κάνει ο προγραμματιστής μπορεί να γίνει λάθος και να εισάγει bugs
  - Λ.Σ.: Πρέπει να διαχειρίζεται την επιστροφή κοινόχρηστων πόρων για να εμποδίσει «σατανικές» συμπεριφορές
- Πώς;
  - Εύκολο αν χρησιμοποιείται σε 1 σημείο: όταν ο ptr εξαφανιστεί, αποδέσμευσε
  - Δύσκολο με διαμοίραση: δε μπορούμε να ανακυκλώσουμε μνήμη έως ότου δεν τη χρειάζεται κανείς

- Διαμοίραση => δείκτες προς τα δεδομένα
  - Ιδέα: όχι δείκτες => ελεύθερο!
- 2 σχήματα: μετρητής αναφορών, mark & sweep garbage collection

Μετρητής αναφορών

- Αλγόριθμος: μέτρα τους δείκτες προς κάθε αντικείμενο
  - Κάθε αντικείμενο έχει ένα μετρητή δεικτών που το δείχνουν

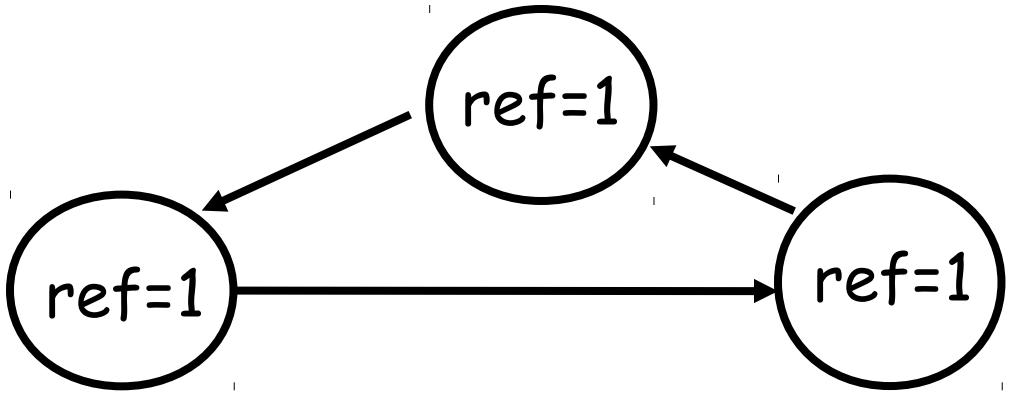
```
void foo(bar c) {
   bar a, b;
   a = c;   <..... c->refcnt++;
   b = a;   <..... a->refcnt++;
   a = 0;   <..... c->refcnt--;
   return;  <..... b->refcnt--;
}
```



  - Μετρητής = 0; Ελευθέρωσε τον πόρο
- Δουλεύει καλά για ιεραρχικές δομές δεδομένων, περιγραφείς αρχείων, σελίδες, κλπ

Προβλήματα

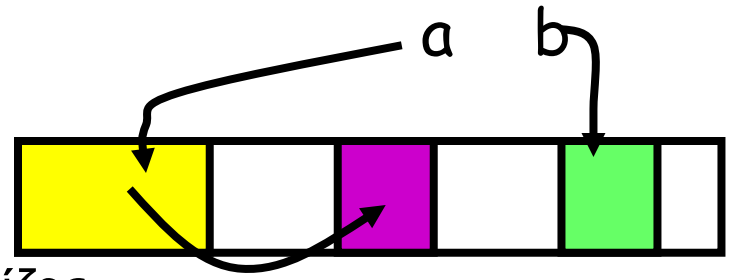
- Κυκλικές δομές δεδομένων: μετρητής > 0
  - Αν δεν υπάρχουν εξωτερικές αναφορές = χαμένο!



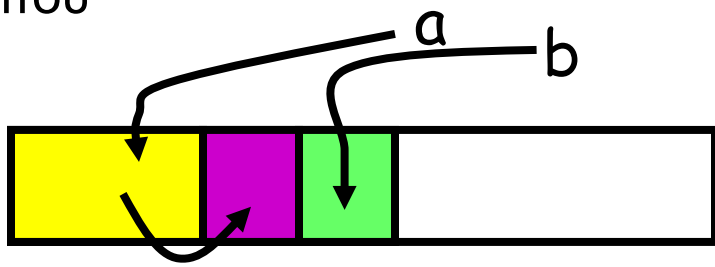
- Απλοϊκό: Πρέπει να «ασχολούμαστε» σε κάθε αναφορά, δημιουργία, καταστροφή αντικειμένου.
- Χωρίς υποστήριξη μεταγλωττιστή εύκολο να ξεχαστεί ένα ++ ή --. Άσχημο bug…

Mark & sweep garbage collection

- Αλγόριθμος: Μάρκαρε τη μνήμη στην οποία μπορούμε να φτάσουμε. Τα υπόλοιπα σκουπίδια…
  - Πρέπει να βρεθούν όλες οι “ρίζες” – Κάθε μεταβλητή global ή stack variable που περιέχει δείκτη προς αντικείμενο
  - Πρέπει να βρεθούν όλοι οι δείκτες μέσα σε αντικείμενα



- 1ο πέρασμα: σημείωση
  - Σημείωσε τη μνήμη που δείχνεται από τις ρίζες. Συνέχισε αναδρομικά σε όλα τα αντικείμενα που δείχνονται από αυτή τη μνήμη …



- 2ο πέρασμα: διαγραφή
  - Διέγραψε τα αντικείμενα που δεν είναι μαρκαρισμένα
  - Μπορείς και να τα μεταφέρεις στο τέλος της heap (συμπίεση) αλλάζοντας τους δείκτες

Λεπτομέρειες

- Μεγάλο πλεονέκτημα: Μπορούμε να αλλάξουμε δείκτες.
  - Άρα και να συμπιέσουμε, αντί να μείνουμε από χώρο
- Βοηθάει η υποστήριξη από τον μεταγλωττιστή (για να γίνουν parse τα αντικείμενα).
  - Java, C#, Modula-3 υποστηρίζουν gc
- Γίνεται και χωρίς υποστήριξη: “συντηρητικό gc”
  - Σε κάθε δέσμευση, κατέγραψε (διεύθυνση, μέγεθος)
  - Σκάναρε τη heap, τα data & τη stack για ακέραιους που θα μπορούσαν να είναι νόμιμες τιμές δεικτών και σημείωσέ τις!