ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Εργαστήριο
Λειτουργικών Συστημάτων
8ο εξάμηνο, Ροή Υ, ΗΜΜΥ

Σχεδιασμός και υλοποίηση υποδομής σημείωσης διεργασιών στον πυρήνα του Linux

Σταυρακάκης Χρήστος, 03106165
Τσιούρης Ιωάννης, 03106193

Η/Μ Παράδοσης: **26 Απριλίου 2010**

Αθήνα, 2010

1 Εισαγωγή

Σκοπός της συγκεκριμένης άσκησης είναι η επίλυση του λόξυγγα ανετοιμότητας 2. Απώτερος σκοπός είναι η εξοικείωση με ένα πραγματικό λειτουργικό σύστημα (Linux) και με τον προγραμματισμό σε επίπεδο πυρήνα. Η αναφορά αυτή αποτελεί μία ενδιάμεση αναφορά που σκοπό έχει να παρουσιάσει της σχεδιαστικές επιλογές για την επίλυση του προβλήματος και *όχι* την υλοποίηση.

Ο προγραμματισμός σε χώρο πυρήνα ενός λειτουργικού συστήματος παρουσιάζει αρκετές δυσκολίες και διαφορές από τον προγραμματισμού σε χώρο χρήστη. Το μέγεθος του πυρήνα καθώς και οι διάφορες απρόβλεπτες εξαρτήσεις μεταξύ τμημάτων κώδικα καθιστούν τον προγραμματισμό πυρήνα μια δύσκολη εργασία. Επίσης παρουσιάζει αρκετές ιδιαιτερότητες όπως η έλλειψη πρόσβασης στη βιβλιοθήκη της C, το μικρό μέγεθος της στοίβας του πυρήνα, η έλλειψη προστασίας μνήμης καθώς και ο ασύγχρονος τρόπος λειτουργίας που καθιστά το συγχρονισμό απαραίτητο. Τέλος στον προγραμματισμό πυρήνα δίνεται μεγάλη έμφαση στην φορητότητα υπό την έννοια, ότι προγράμματα που είναι ανεξάρτητα συγκεκριμένης αρχιτεκτονικής να μπορούν να μεταγλωττιστούν και να εκτελεστούν σε διαφορετικά συστήματα.

2 Περιγραφή του προβλήματος

Ο λόξυγγας ανετοιμότητας αναφέρεται στο φαινόμενο της επιβράδυνσης της εκτέλεσης μιας λειτουργίας σε σχέση με προηγούμενες εκτελέσεις της λόγω της απώλειας από την κύρια μνήμη των απαραίτητων δεδομένων για την λειτουργία της. Η απώλεια αυτή οφείλεται είτε στην αίτηση μνήμης από κάποια άλλη διεργασία είτε στην δραστηριότητα κάποιας διεργασίας σε αρχεία.

Θέλουμε να αποτρέψουμε το φαινόμενο η δραστηριότητα σε κάποια αρχεία να εκτοπίζει χρήσιμα ”παλιά” αρχεία που είναι cacheαρισμένα στη μνήμη. Η επιλογή αυτή βασίζεται στο γεγονός ότι σε πολλές περιπτώσεις η διατήρηση των αρχείων στη μνήμη cache δεν είναι απαραίτητη. Για παράδειγμα η αντιγραφή ενός αρχείου (χωρίς περαιτέρω, χρήση του αρχείου) που βασίζεται σε μία σειριακή προσπέλαση αυτού, δεν έχει κάποιο λόγω να διατηρεί το αρχείο στη μνήμη.

Για να το επιτύχουμε αυτό, πρέπει να ”σημειώσουμε” ποιες διεργασίες δεν χρειάζεται να διατηρούν τα αρχεία τους στη μνήμη. Αυτή η σημείωση θα είναι η κατάρα `no-fs-cache` η περιγραφή της οποίας ακολουθεί (βλ. παρ. 4).

Θα δημιουργήσουμε επίσης μία γενικότερη υποδομή που θα επιτρέπει στο χρήστη να ”σημειώνει” διεργασίες με διαφορές κατάρες. Η υποδομή αυτή είναι η `curse` και η περιγραφή της υπάρχει στη συνέχεια (παρ. 3).

3 Υποδομή Curse

Όπως ήδη αναφέραμε η υποδομή αυτή θα μας επιτρέπει να ”σημειώνουμε” διεργασίες με κατάρες. Η υποδομή αυτή θα αποτελείται από τα εξής στοιχεία :

- Κλήση συστήματος
- Στατική βιβλιοθήκη χρήστη
- Εργαλείο γραμμής εντολών
- Υλοποίηση λειτουργιών και προγραμματιστική διεπαφή.

Στην αναφορά αυτή δεν θα ασχοληθούμε με λεπτομέρειες υλοποίησης.

3.1 Why System Call

Το βασικό κομμάτι της υποδομής curse θα αποτελεί κώδικα πυρήνα, καθώς είναι απαραίτητη η χρήση βασικών δομών του λειτουργικού συστήματος. Για να μπορεί ο χρήστης να επεξεργαστεί αυτές τις πληροφορίες είναι απαραίτητη η υλοποίηση μιας κλήσης συστήματος. Μέσω αυτής μπορεί να εκτελέσει κώδικα σε χώρο πυρήνα (Kernel Space).Το system call που θα υλοποιήσουμε είναι το `sys_curse`.

3.2 Προσθήκη System Call

Για την δημιουργία του system call θα ακολουθήσουμε τα εξής βήματα:

1. Προσθέτουμε στο αρχείο `arch/x86/include/asm/unistd_32.h` τον ορισμό του αριθμού κλήσης της `sys_curse`. Αυτός είναι κατά ένα μεγαλύτερος από τον τελευταίο ήδη ορισμένο αριθμό. Επίσης πρέπει να αυξήσουμε κατά ένα τον αριθμό της σταθεράς `NR_syscalls` που δηλώνει τον αριθμό των ορισμένων system calls.

2. Προσθέτουμε στο αρχείο `arch/x86/kernel/syscall_table_32.S`, το οποίο αποτελεί τον πίνακα των κλήσεων συστήματος την συνάρτηση που θα υλοποιεί την κλήση συστήματος που προσθέσαμε. πχ `.long sys_curse`.

3. Προσθέτουμε στο αρχείο `include/linux/syscalls.h` την δήλωση της συνάρτησης που θα την υλοποιεί : `asmlinkage long sys_curse(int flag, short int curse ,pid_t pid,char **list_curses);`

4. Τέλος πρέπει να προσθέσουμε το αρχείο με την υλοποίηση του system call μας. Το αρχείο αυτό θα είναι το curse.c και θα περιέχεται στο φάκελο kernel/. Θα πρέπει να ενημερώσουμε και το Makefile ώστε να γίνεται η μεταγλώττιση του συγκεκριμένου αρχείου.

3.3 Απαραίτητες Δομές

Για την λειτουργία της υποδομής curse είναι απαραίτητο να γίνει η προσθήκη κάποιων δομών οι οποίες θα περιέχουν απαραίτητες πληροφορίες. Αρχικά απαιτείται πληροφορία για το ποιες κατάρες είναι διαθέσιμες στο σύστημα και ποιες ενεργοποιημένες. Αυτές οι πληροφορίες θα αποθηκευτούν στο αρχείο `curse.h`

Η κάθε κατάρα μοντελοποιείται ως ένα αναμμένο bit σε ένα 16-bit αριθμό.

```
#DEFINE NO_FS_CACHE 0x01
#DEFINE NO_EXEC 0x02
#DEFINE POISON 0x04
```

Η δομή που θα επιλέξουμε για να αποθηκεύσουμε τα ονόματα των διαθέσιμων κατάρων στο σύστημα θα είναι στατική, ένας πίνακας από strings. Η θέση στον πίνακα όπου βρίσκεται το όνομα κάθε κατάρας αντιστοιχεί στη θέση όπου η συγκεκριμένη κατάρα έχει 1. Κάθε αλφαριθμητικό στον πίνακα έχει μέγεθος 32 bytes. Η πληροφορία αυτή βρίσκεται στο χώρο πυρήνα ώστε το ίδιο εκτελέσιμο να μπορεί να χρησιμοποιηθεί σε συστήματα με διαφορετικές κατάρες.

```
#DEFINE CURSE_NAME_LENGTH 32 * sizeof(char) ;
const char * curses[] = {"no_fs_cache","no_exec","poison"} ;
```

Ο αριθμός των κατάρων που βρίσκονται στο σύστημα είναι

```
#DEFINE NR_CURSES 3
```

Επίσης θα πρέπει να υπάρχει η πληροφορία για το ποιες κατάρες είναι ενεργοποιημένες και ποιες απενεργοποιημένες. Για το σκοπό αυτό θα εισάγουμε μία νέα global μεταβλητή τύπου short int (16 bits). Τη μεταβλητή αυτή θα τη χρησιμοποιήσουμε σε επίπεδο bit. Αν το x bit είναι 1, σημαίνει ότι η κατάρα x είναι ενεργοποιημένη. Θα ονομάσουμε αυτή τη μεταβλητή `curses_enabled`. Στην αρχή όλες οι κατάρες είναι απενεργοποιημένες.

```
short int cursed_enabled = 0x00
```

Σε επίπεδο διεργασίας θα χρειαστούμε μία μεταβλητή ανά διεργασία η οποία θα δείχνει ποιες κατάρες έχει υποστεί μια διεργασία. Η μεταβλητή αυτή θα είναι πάλι τύπου short int, με την ίδια λογική της `cursed_enabled` και θα ονομάζεται `cursed`. Η μεταβλητή αυτή ενσωματώνεται μέσα στη δομή `struct task_struct` η οποία υπάρχει στο αρχείο `include/linux/sched.c`. Η δομή αυτή αποτελεί τον process descriptor της διεργασίας, και περιέχει όλες τις απαραίτητες πληροφορίες που διατηρεί το λειτουργικό σύστημα για μια διεργασία, όπως το pid της διεργασίας, την κατάσταση της, vm state, ανοιχτά αρχεία, credentials κτλπ..

3.4 Λειτουργία System Call Curse

Το system call θα πρέπει να υποστηρίζει 7 διαφορετικές λειτουργίες, οι οποίες αναλύονται παρακάτω. Ο διαχωρισμός για το ποια λειτουργία θα εκτελεστεί θα βασίζεται στο flag το οποίο θα περνιέται ως πρώτο όρισμα. Για ευκολία ανάγνωσης κώδικα τα flag θα είναι pre-defined σταθερές.

- **Αίτηση λίστας με διαθέσιμες κατάρες** Στην περίπτωση αυτή θέλουμε να επιστρέψουμε στο χρήστη τον πίνακα με τα ονόματα των διαθέσιμων κατάρων. Ο πίνακας όμως βρίσκεται σε χώρο πυρήνα, οπότε δε μπορούν να μοιραστούν άμεσα τη δομή. Για το λόγο αυτό θα χρησιμοποιήσουμε τη συνάρτηση `copy_to_user` που χρησιμοποιείται για τη μεταφορά δεδομένων από χώρο πυρήνα σε χώρο χρήστη. Ο χρήστης θα κάνει malloc τον κατάλληλο πίνακα για να δεχθεί τις διαθέσιμες κατάρες και θα περνάει τη διευθυνσή του ως το τέταρτο όρισμα της `sys_curse`, το `list_curses`.

- **Ενεργοποίηση κατάρας**
  Για την ενεργοποίηση μιας κατάρας θα περνιέται ως δεύτερο όρισμα ο αριθμός που αντιστοιχεί στην κατάρα που θέλουμε να ενεργοποιήσουμε. Η ενεργοποίηση θα εκτελείται με το λογικό and (||) μεταξύ της σταθεράς που αντιστοιχεί στην κατάρα και της μεταβλητής `cursed_enabled`.

  Ωστόσο, εδώ προκύπτουν κάποια ζητήματα :

  - Συγχρονισμός : Είναι πιθανόν να παρουσιαστούν race conditions που αφορούν την πρόσβαση στη μεταβλητή `cursed_enabled`. Για να λυθούν αυτή το πρόβλημα θα χρησιμοποιήσουμε κάποιο σχήμα συγχρονισμού. Καθώς η πρόσβαση στη μεταβλητή `cursed_enabled`, αφορά μόνο την ενεργοποίηση ενός bit της, μπορούμε να χρησιμοποιήσουμε κάποια ατομική εντολή (atomic operation).
    Αντί λοιπόν για το λογικό or, Θα χρησιμοποιήσουμε λοιπόν την ατομική εντολή `set_bit(int nr,void *addr)`, η οποία θέτει το nr bit της μεταβλητής στην οποία δείχνει ο pointer addr.
  - Ασφάλεια : Καθώς όλοι οι χρήστες δεν έχουν το δικαίωμα να ενεργοποιούν και απενεργοποιούν κατάρες, είναι απαραίτητο να γίνεται κάποιος έλεγχος δικαιωμάτων. Την ενεργοποίηση κατάρων μπορεί να πραγματοποιήσει μόνο ο root, ο έλεγχος αυτός γίνεται με τη συνάρτηση του πυρήνα `capable(CAP_SYS_ADMIN)`.

- **Απενεργοποίηση κατάρας**
  Για την απενεργοποίηση κατάρας ισχύει ότι για την ενεργοποίηση, με τη διαφορά ότι αντί της `set_bit` θα χρησιμοποιηθεί η `clear_bit`.

- **Ερώτηση κατάστασης ενεργοποίησης κατάρας**
  Η ερώτηση κατάστασης ενεργοποίησης κατάρας πραγματοποιείται με την επιλογή του κατάλληλου flag καθώς και του αριθμού κατάρας. Ο έλεγχος γίνεται ελέγχοντας αν το αντίστοιχο bit της μεταβλητής `cursed_enabled` είναι 0 ή 1. Άρα ισχύει ότι για την ενεργοποίηση/απενεργοποίηση, χωρίς τον έλεγχο δικαιωμάτων και χρησιμοποιείται η ατομική εντολή `test_bit`.

- **Επιβολή κατάρας σε διεργασία**
  Για την επιβολή κατάρας απαιτείται το κατάλληλο flag, ο αριθμός κατάρας καθώς και το `pid` της διεργασίας που θέλουμε να καταραστούμε. Η επιβολή κατάρας σε διεργασία θυμίζει πολύ την ενεργοποίηση κατάρας. Όμως σε αυτήν την περίπτωση το λογικό or θα γίνει μεταξύ του ειδικού αριθμού κατάρας, και της μεταβλητής `cursed` η οποία υπάρχει στην δομή task_struct που αντιστοιχεί στη διεργασία με το αντίστοιχο pid.

  Για να βρούμε το `task_struct` που αντιστοιχεί στη διεργασία με το αντίστοιχο pid θα χρησιμοποίησουμε την `struct *task_struct *find_task_by_vpid(pid_t vnr)`. Αυτή θα μας επιστρέψει τον κατάλληλο process descriptor.. Για να χρησιμοποίησουμε αυτή την εντολή, θα χρειαστεί να "πάρουμε" κάποιο lock, το όποια επιτυγχάνουμε με τις εντολές `rcu_read_lock()` και `rcu_read_unlock()`.

  Και σ' αυτήν την περίπτωση ενδέχεται να παρουσιαστούν race conditions για την μεταβλητή `cursed`. Οπότε θα χρησιμοποιήσουμε και εδώ ατομική εντολή, και συγκεκριμένα την `set_bit`.

  Ακόμα, οι μόνοι χρήστες που έχουν δικαίωμα επιβολής μιας κατάρας σε διεργασία είναι ο ιδιοκτήτης της διεργασίας και ο root. Θα πρέπει λοιπόν να γίνει κάποιος έλεγχος για το αν τα credentials του χρήστη που επιβάλει την διεργασία είναι ίδια με τα credential της διεργασίας που θέλει να καταραστεί. Το uid μαζί με τα υπόλοιπα credentials υπάρχει στη `struct cred` που υπάρχει στο `task_struct`. Το `task_struct` της διεργασίας που πρόκειται να καταραστεί έχει βρεθεί ήδη με βάση το pid. Το `task_struct` της διεργασίας που επιχειρεί την κατάρα υπάρχει σε μία μεταβλητή του συστήματος που δείχνει την τρέχουσα διεργασία, την `current`. Το cred της μπορούμε να το πάρουμε εύκολα με την μακροεντολή `current_cred()`. Ο έλεγχος θα γίνει με τις παρακάτω εντολές , όπου το tcred θα έχει τα credentials της τρέχουσας διεργασίας και το cred της διεργασίας που θέλουμε να καταραστούμε.

```
    (cred->euid  ^ tcred->suid) &&
    (cred->euid  ^ tcred->uid) &&
    (cred->uid   ^ tcred->suid) &&
    (cred->uid   ^ tcred->uid)  ||
    (capable(CAP\_SYS\_ADMIN)
```

  Μία από τις απαιτήσεις της υποδομής είναι οι κατάρες να κληρονομούνται στα παιδιά της διεργασίας ενώ να μην επηρεάζονται τα ήδη υπάρχοντα παιδιά τους. Η δημιουργία ενός παιδιού γίνεται με χρήση της fork η οποία ορίζεται στο αρχείο `kernel/fork.c`. Η `fork` καλεί την `clone`, η οποία καλεί την `do_fork` η οποία καλεί την `copy_process`. Η τελευταία δημιουργεί ένα νέα kernel stack, thread_info και task_struct για τη νέα διεργασία. Οι τιμές που παίρνουν είναι πανομοιότυπες με αυτές του πατέρα, με μοναδική εξαίρεση το pid. Για να επιτύχουμε λοιπόν την κληρονομικότητα αρκεί να φροντίσουμε η `copy_process` να αντιγράφει και την τιμή της μεταβλητής `cursed`.

- **Αφαίρεση κατάρας σε διεργασία**
  Ισχύουν τα ίδια με την επιβολή κατάρας, με τη διαφορά ότι χρησιμοποιείται η ατομική εντολή `clear_bit`.

- **Ερώτηση καθεστώτος κατάρας σε διεργασία**
  Ισχύουν τα ίδια με την επιβολή/αφαίρεση κατάρας, με τη διαφορά ότι χρησιμοποιείται η ατομική εντολή `test_bit`.

Σημειώνουμε ότι σε κάθε περίπτωση, η κλήση συστήματος θα επιστρέφει κάποιον κωδικό επιτυχίας αν η λειτουργία ήταν επιτυχής, η τον κατάλληλο κωδικό λάθους.

3.5 Βιβλιοθήκη Χρήστη

Η βιβλιοθήκη αυτή θα παρέχει στο χρήστη μια διεπαφή για τα προγράμματα χρήστη. Σκοπός της είναι να παρέχει στους χρήστες ένα εύκολο τρόπο να χρησιμοποιούν τις λειτουργίες της δομής curse χωρίς να χρειάζεται κάθε φορά να κάνουν την κατάλληλη κλήση στη κλήση συστήματος `sys_curse` και να χρειάζεται να διαμορφώνουν όλα τα ορίσματα της. Αυτό θα απαιτούσε από το χρήστη να θυμάται τα κατάλληλα flags, τον αριθμό της κλήσης συστήματος κτλπ. Επίσης καθώς κάποια από τα ορίσματα της `sys_curse` δεν χρησιμοποιούνται σε ορισμένες λειτουργίες θα έπρεπε ο χρήστης να θυμάται ποια είναι και να τα θέτει κάθε φορά σε `NULL`. Είναι εμφανές ότι αυτή η προσέγγιση εκτός από μη φιλική είναι και επιρρεπής σε λάθη του χρήστη.

Για να αποφύγουμε λοιπόν όλα αυτά δημιουργούμε τη βιβλιοθήκη χρήστη που ουσιαστικά θα αποτελείται από wrapper συναρτήσεις για την κλήση της `sys_curse`. Η βιβλιοθήκη θα περιλαμβάνει τουλάχιστον τόσες συναρτήσεις όσες οι λειτουργίες που υποστηρίζει το system call. Αυτές οι συναρτήσεις θα είναι υπεύθυνες και για την αναγνώριση των κωδικών λαθών που επιστρέφει το system call και την πιο φιλικά παρουσίαση τους στο χρήστη. Η βιβλιοθήκη αυτή μπορεί να υπάρχει σε οποιοδήποτε σημείο έτσι ώστε να μπορεί να γίνει include από το χρήστη.

3.6 Εργαλείο Γραμμής Εντολών

Τέλος η υποδομή curse περιλαμβάνει ένα εργαλείο γραμμής εντολών. Σκοπός αυτού του εργαλείου είναι η εύκολη αλληλεπίδραση με τις λειτουργίες της curse.

Αυτό το εργαλείο μπορεί ουσιαστικά να είναι ένα ψευδο-φλοιός(shell). Ο χρήστης θα δίνει μέσω αυτού ότι εντολές επιθυμεί. Επίσης ο χρήστης μπορεί να καταριέται αυτόν τον φλοιό και μέσω αυτού να τρέχει όσες διεργασίες θέλει. Λόγω κληρονομικότητας οι διεργασίες που θα τρέχει από εκεί θα είναι και αυτές καταραμένες.

3.7 Σχολιασμός

Η υποδομή curse είναι ο βασικός μηχανισμός υλοποίησης και χρήσης κατάρων στο λειτουργικό σύστημα. Ο σχεδιασμός του έγινε με γνώμονα την επεκτασιμότητα και ανεξαρτησία συστήματος. Η υλοποίηση μιας νέας κατάρας καθίσταται πολύ εύκολη καθώς ο χρήστης αρκεί να κάνει `define` τον αριθμό της κατάρας που θέλει να υλοποιήσει, να αυξήσει κατά ένα τον γενικό αριθμό κατάρων,να προσθέσει το όνομα της κατάρας και φυσικά να υλοποιήσει την κατάρα του. Δεν χρειάζεται να γράψει επιπλέον κώδικα για την υποδομή curse ούτε να πειράξει κάποια από τις δομές τις. Όσον αφορά την ανεξαρτησία συστήματος και υλοποιημένων κατάρων, η υποδομή curse είναι υλοποιημένη κατά τέτοιο τρόπο, ώστε το ίδιο εκτελέσιμο να λειτουργεί σε συστήματα με διαφορετικές κατάρες. Αυτό επιτυγχάνεται καθώς όλη η πληροφορία για το ποιες κατάρες είναι υλοποιημένες υπάρχει αποκλειστικά στον χώρο πυρήνα και όχι στον χώρο χρήστη.

Καθώς η υποδομή curse ”λειτουργεί” σε χώρο πυρήνα, έπρεπε να δοθεί ιδιαίτερη έμφαση σε θέματα όπως της ασφάλειας και του συγχρονισμού. Τα ζητήματα αυτά λύθηκαν με τους κατάλληλους ελέγχους δικαιωμάτων και σχημάτων συγχρονισμού. Όσων αφορά τα δεύτερα, υπήρχαν πολλές επιλογές, όπως semaphores, spinlocks και atomic transactions. Θα μπορούσε να χρησιμοποιηθεί το καθένα από αυτά, όμως καθώς η φύση των λειτουργιών το επέτρεπε χρησιμοποιήθηκαν οι ατομικές εντολές οι οποίες είναι πιο αποδοτικές.

Όσων αφορά την κληρονομικότητα των κατάρων, ήταν ο μηχανισμός δημιουργίας διεργασιών στο λειτουργικό σύστημα που ουσιαστικά έλυνε το πρόβλημα.

Είναι λογικό ότι η παρεμβάσεις στο λειτουργικό σύστημα θα προσθέτουν κάποια επιβάρυνση. Σκοπός μας ήταν η ελαχιστοποίηση αυτών των επιβαρύνσεων. Αρχικά φροντίσαμε την χρήση όσων των δυνατών λιγότερων μεταβλητών ώστε να μην επιβαρύνουμε την ήδη μικρή kernel stack. Έτσι οι ενεργοποιημένες κατάρες μοντελοποιήθηκαν με μία μόνο μεταβλητή και μάλιστα τύπου short int. Μια μικρή επιβάρυνση υπάρχει στο ότι προστίθεται μία μεταβλητή ένα διεργασία, όμως δεδομένου ότι το ήδη μέγεθος της δομής `task_struct` είναι ήδη περίπου $1.7 kilobytes$, η προσθήκη μιας short int δεν θεωρείται ιδιαίτερη επιβαρυντική.

Ούτε από άποψη χρόνου υπάρχει ιδιαίτερη επιβάρυνση. Οι περισσότερες λειτουργίες αντιστοιχούν στην αλλαγή η έλεγχο ενός bit μιας μεταβλητής. Η μόνη σημαντική επιβάρυνση προκύπτει από το γεγονός ότι το λειτουργικό σύστημα για να ελέγχει αν μια κατάρα είναι ενεργοποιημένη σε μια διεργασία θα πρέπει να κάνει τον έλεγχο αν το αντίστοιχο bit της μεταβλητής `cursed_enabled` και της μεταβλητής της διεργασίας `cursed` είναι και τα δύο 1.

Τέλος πρέπει να σημειώσουμε ότι αν ο τύπος της κατάρας είναι τέτοιος όπου το λειτουργικό σύστημα δεν εμπλέκεται συχνά για να ελέγξει τις δύο μεταβλητές, μπορεί η ενεργοποίηση μιας κατάρας ή η επιβολή κατάρας σε μια διεργασία να αργήσει να γίνει αντιληπτή.

4 Κατάρα no-fs-cache

Στο σημείο αυτό θα ασχοληθούμε με τον σχεδιασμό της κατάρας `no-fs-cache` η οποία θα λύσει το αρχικό μας πρόβλημα, δηλαδή τον λόξυγγα ανετοιμότητας. Η κατάρα αυτή θα ενσωματωθεί στην γενικότερη υποδομή για της κατάρες, δηλαδή την `curse`.

Όπως ήδη σημειώσαμε σκοπός μας είναι συγκεκριμένες διεργασίες οι οποίες δεν χρειάζεται να διατηρούν τα δεδομένα τους στην cache να μην εκτοπίζουν δεδομένα από τις διεργασίες οι οποίες τα χρειάζονται. Καθώς όμως το λειτουργικό σύστημα δε γνωρίζει ποια πρόκειται να είναι η χρήση των αρχείων, αυτή η λειτουργία δεν μπορεί να αυτοματοποιηθεί. Η κατάρα μας λοιπόν είναι ένας τρόπος ώστε να ενημερώνουμε το λειτουργικό σύστημα για την προβλεπόμενη χρήση των αρχείων.

Για την υλοποίηση της κατάρας μας θα χρησιμοποιήσουμε την κλήση της fadvise η οποία υπάρχει στο πρότυπο POSIX.

```
int posix_fadvise(int fd, off_t offset, off_t len, int advice);
```

Αυτή χρησιμοποιείται για να δηλώσει στο λειτουργικό σύστημα ποια χρήση πρόκειται να έχει ένα αρχείο, και υποστηρίζει κάποιες επιλογές όπως `NORMAL, SEQUENTIAL, RANDOM, NOREUSE, WILLNEED DONTNEED`, παρότι η επιλογή NOREUSE που μοιάζει να είναι αυτή η οποία ταιριάζει στην περίπτωση μας δεν είναι υλοποιημένη. [1] Θα χρησιμοποιήσουμε λοιπόν την επιλογή DONTNEED. Η συμπεριφορά του linux σε αυτή την επιλογή είναι να αποβάλλει από τη μνήμη cache τα περιεχόμενα του αρχείου που δείχνει ο file descriptor `fd`. Το ποια δεδομένα θα αποβληθούν από τη μνήμη ορίζεται από το συνδυασμό `offset` και `length`.

Θα χρειαστεί λοιπόν να ενσωματώσουμε κάποιες κλήσεις της fadvise σε κλήσεις συστήματος του linux. Σύμφωνα με λειτουργιά της fadvise στην επιλογή `DONTNEED`, θα πρέπει κάποιο μέρος του

[1] mm/fadvice.c : lines 108,109: `case POSIX_FADVISE_NOREUSE: break;`

αρχείου να έχει ήδη μεταφερθεί στη μνήμη και από εκεί να το αποβάλλουμε. Οι κλήσεις λοιπόν της fadvise θα ενσωματωθούν στις κλήσεις συστήματος `read` και `write` οι οποίες ορίζονται στο αρχείο `fs/read_write.c`.

Θα περιγράψουμε την κλήση συστήματος read, η οποία ορίζεται στο αρχείο `fs/read_write.c` στην γραμμή 372:

```
SYSCALL_DEFINE3(read, unsigned int, fd, char __user *, buf, size_t,
    count)
```

Η συνάρτηση αυτή δέχεται τον file descriptor του αρχείου, τον buffer στον οποίο θα γράψει τα δεδομένα για να τα επιστρέψει στο χρήστη και τέλος το πόσο θέλουμε να διαβάσουμε.Αρχικά ψάχνει στην δομή `struct_file (include/linux/fs.h)` για να βρει τον file pointer(`loff_t f_pos`) μέσω της `file_pos_read`. Στην συνέχεια διαβάζει `count` bytes από το αρχείο και τα αποθηκεύει στον buffer. Τέλος σώζει μέσω της `file_pos_write` τη νέα τιμή του file pointer. Σε αυτό το σημείο λοιπόν τα δεδομένα που θέλουμε να διαβάσουμε έχουν αποθηκευτεί στο buffer και η δομή `struct_file` είναι συνεπής.

Σε ακριβώς αυτό το σημείο θα γίνεται η κλήση της fadvise για τις καταραμένες διεργασίες. Ο έλεγχος για το αν μια διεργασία είναι καταραμένη θα γίνεται ελέγχοντας ότι το bit που αντιστοιχεί στην συγκεκριμένη κατάρα θα είναι ενεργοποιημένο (λογικό 1) τόσο στη μεταβλητή `curse_enabled` (γενική ενεργοποίηση κατάρας), όσο και στη μεταβλητή `cursed` της διεργασίας που έκανε το read. Αν αυτή η συνθήκη ισχύει τότε θα γίνεται κλήση της fadvise με τα εξής ορίσματα :

- File Descriptor : Τον file descriptor της read
- Offset : Την τιμή του file pointer(loff_t tpos) μείον την τιμή των bytes που διαβάσαμε. Δηλαδή pos-count.
- Length : Την τιμή του count.

Συνεπώς μετά από κάθε διάβασμα θα αποβάλουμε από τη μνήμη τα δεδομένα τα οποία διαβάσαμε.

Ακριβώς η ίδια προσέγγιση θα χρησιμοποιηθεί και για την write.

4.1 Ένταξη κατάρας στην υποδομή curse

Όπως αναφέραμε η κατάρα αυτή θα ενταχθεί στην γενικότερη υποδομή κατάρων curse. Ο τρόπος ένταξης κατάρας έχει ήδη αναλυθεί, αλλά επισημαίνουμε εδώ την ένταξη της συγκεκριμένης κατάρας. Η κατάρα αυτή είναι η πρώτη που εντάσσουμε άρα θα χρησιμοποιήσουμε το πρώτο bit των μεταβλητών `cursed_enabled` και `cursed`.

Στο αρχείο `curse.h` κάνουμε τα εξής define:

```
#DEFINE NO_FS_CACHE 0x01
#DEFINE NR_CURSES 1
```

Επίσης προσθέτουμε το όνομα της κατάρας στην πρώτη θέση του πίνακα curses:

```
const char *curses[] = {"no_fs_cache"};
```

Οπότε είναι το πρώτο bit των μεταβλητών `cursed_enabled` και `cursed` όπου θα ελέγχουμε για να κάνουμε την κλήση της fadvise.

4.2 Σχολιασμός

Με τον συγκεκριμένο σχεδιασμό καταφέρνουμε να αντιμετωπίσουμε τον λόξυγγα ανετοιμότητας. Ωστόσο περιμένουμε ο συγκεκριμένος σχεδιασμός να έχει κάποια αρνητική επίδραση στην επίδοση του συστήματος. Αυτό οφείλεται στο γεγονός ότι σε κάθε read ή write που πραγματοποιείται θα πρέπει να γίνεται ένας έλεγχος για το αν η διεργασία είναι καταραμένη και επίσης να γίνεται και μια κλήση fadvise όταν όντος είναι. Καθώς όμως οι λειτουργίες ανάγνωσης και εγγραφής είναι πολύ συχνές στα περισσότερα υπολογιστικά συστήματα, η προσθήκη αυτού του ελέγχου μπορεί τελικά να θεωρηθεί ”ακριβή”.

Όσον αφορά τη σταθερότητα και ασφάλεια του συστήματος, θεωρούμε ότι ο σχεδιασμός μας είναι αρκετά επιτυχημένος. Αυτό βασίζεται στο γεγονός ότι η κατάρα no-fs-cache δεν τροποποιεί υπάρχουσες δομές του λειτουργικού μας συστήματος, ούτε επιτελεί ”επικίνδυνες” λειτουργίες σε αυτές. Αντίθετα βασίζεται στην υποδομή curse και στον τρόπο που το λειτουργικό σύστημα εκτελεί τις λειτουργίες ανάγνωσης και εγγραφής καθώς και τον τρόπο που μεταφέρει και απομακρύνει αρχεία από τη μνήμη cache. Καθώς αυτή η εργασία δεν αποτελεί παρά έναν σχεδιασμό της υλοποίησης ελπίζουμε ότι τα συμπεράσματα μας είναι σωστά και ότι θα επαληθευτούν στο στάδιο της υλοποίησης.