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

Τελική αναφορά

Σταυρακάκης Χρήστος, 03106165
Τσιούρης Ιωάννης, 03106193

Η/Μ Παράδοσης: **10 Ιουνίου 2010**

Αθήνα, 2010

1 Εισαγωγή

Σκοπός αυτής της αναφοράς είναι η παρουσίαση της τελικής σχεδίασης της υποδομής σημείωσης διεργασιών με κατάρες στον πυρήνα του linux καθώς και της κατάρας no-fs-cache. Στην αναφορά αυτή δεν θα δοθεί έμφαση στην ανάλυση του προβλήματος ή στις σχεδιαστικές επιλογές, καθώς αυτό έγινε αναλυτικά στην ενδιάμεση αναφορά. Ωστόσο, θα εστιάσουμε στα σημεία στα οποία η υλοποίηση μας δεν ακολούθησε την ενδιάμεση σχεδίαση. Τέλος θα παρουσιαστεί η τεκμηρίωση για τους χρήστες.

2 Τελική σχεδίαση

2.1 Υποδομή σημείωσης διεργασιών

Η σημείωση διεργασιών στον πυρήνα του linux γίνεται μέσω ενός system call, του `sys_curse` το οποίο υλοποιείται και δηλώνεται στον πυρήνα του linux.

Τα ονόματα των κατάρων υπάρχουν αποκλειστικά στον πυρήνα. Η δομή `struct curses_names` περιέχει μια μεταβλητή που δείχνει τον αριθμό των υλοποιημένων κατάρων και έναν πίνακα(names) από pointer σε string με τα ονόματα τους.

```
typedef struct {
    int nr_names;
    const char *names[MAX_NAME_LIST_NAME_LEN];
} name_list_t ;

static name_list_t curses_names = {
                        .nr_names = 2,
                        .names = { [CURSE_STINK]    = "stink",
                                   [CURSE_NOCACHE] = "nocache" }
};
```

Τα ονόματα αυτά πρέπει να μπορούν να ανακτηθούν από το χρήστη. Καθώς όμως ο πίνακας βρίσκεται σε χώρο πυρήνα ο χρήστης δε μπορεί να τον προσπελάσει. Απαιτείται λοιπόν να αντιγραφεί στο χώρο χρήστη. Αυτό επιτυγχάνεται με τη συνάρτηση `copy_to_user` η οποία πραγματοποιεί την αντιγραφή έπειτα από κάποιος απαραίτητους ελέγχους. Όμως η αντιγραφή ενός πίνακα από pointer σε string απαιτεί την αντιγραφή του κάθε στοιχείου ξεχωριστά. Για να αποφευχθεί αυτό επιλέχθηκε η συνένωση των ονομάτων των κατάρων σε ένα string, διαχωρισμένα με κάποιο ειδικό στοιχείο (separator="|"). Το string αυτό γίνεται εύκολα copy to user, και έπειτα ο χρήστης το χειρίζεται κατάλληλα.

Στον πυρήνα μας ορίζουμε τη μεταβλητή `curses_status` η οποία θα χρησιμοποιηθεί για την αποθήκευση της κατάστασης (ενεργοποιημένη ή όχι) των κατάρων. Η μεταβλητή αυτή θα χρησιμοποιηθεί σαν Bitmap. Δηλαδή κάθε κατάρα θα γίνεται map σε κάποιο από τα bit της μεταβλητής `curses_status`.Το ποιο bit αντιστοιχεί σε ποια κατάρα γίνεται define. Αν το bit είναι 1 η κατάρα είναι ενεργοποιημένη, ενώ αν είναι 0 η κατάρα είναι απενεργοποιημένη.

```
static unsigned long curses_status = 0x0;
```

Καθώς όμως ο χρήστης δίνει όνομα κατάρας, χρειάζεται κάποιος τρόπος αντιστοίχισης του ονόματος σε bit. Αυτό επιτυγχάνεται με τη βοήθεια του πίνακα curses_names.names. Στον πίνακα αυτό, τα ονόματα δηλώνονται με τέτοιο τρόπο ώστε το όνομα της κατάρας που αντιστοιχεί στο x bit να βρίσκεται στην x θέση του πίνακα(zero indexed). Για να βρούμε λοιπόν το ποιο bit αντιστοιχεί σε ποια κατάρα, αρκεί μια γραμμική αναζήτηση στον πίνακα. Η γραμμική αναζήτηση γενικά

αυξάνει την πολυπλοκότητα. Όμως καθώς ο αριθμός των κατάρων είναι μικρός δεν δημιουργείται πρόβλημα.

Όσον λοιπόν αφορά της λειτουργίες ελέγχου κατάστασης κατάρας, ενεργοποίησης και απενεργοποιήσης αρκεί ο έλεγχος ή η μεταβολή του bit της μεταβλητής `curses_status` που αντιστοιχεί στην κατάρα. Τα race conditions που προκύπτουν για τον έλεγχο και μεταβολή αυτής της μεταβλητής λύνονται με χρήση bitwise atomic operations. Συγκεκριμένα χρησιμοποιούνται οι `test_bit`, `set_bit` και `clear_bit`.

Αντίστοιχη σχεδίαση χρησιμοποιείται και για τον αν μια διεργασία είναι καταραμένη ή όχι. Σε αντιστοιχία με την μεταβλητή `curses_status` χρησιμοποιείται μια μεταβλητή ανά διεργασία η οποία αποθηκεύεται μέσα στη δομή `struct task_struct`, που αποτελεί τον proccess descriptor της διεργασίας. Η μεταβλητή αυτή είναι η `curses`.

```
stuct task_struct{
volatile long state;
. . .
unsigned long curses;
....
}
```

Όταν λοιπόν θέλουμε να ελέγξουμε αν μια διεργασία είναι καταραμένη, να την καταραστούμε ή να τις αφαιρέσουμε την κατάρα, αρκεί να ελέγξουμε η να μεταβάλουμε το αντίστοιχο bit.

Ο χρήστης λοιπόν δίνει το pid της διεργασίες που τον ενδιαφέρει και εμείς πρέπει να βρούμε το `task_struct` της και να χρησιμοποιήσουμε την μεταβλητή `curses`. Για να βρούμε το task_struct της διεργασίας με βάση το pid χρησιμοποιούμε τη συνάρτηση `find_task_by_vpid(pid)`. Ο έλεγχος και η μεταβολή της μεταβλητής `curses` γίνεται με atomic operations κατά αντιστοιχία με τη global μεταβλητή `curses_status`. Για την εύρεση του task_struct καθώς και τον έλεγχο της μεταβλητής προκύπτουν ζητήματα ταυτόχρονης πρόσβασης και συγχρονισμού. Τα ζητήματα αυτά λύνονται με το lock του πυρήνα του linux `tasklist_lock`. Αυτό το lock αφορά την λίστα με όλα τα `task_struct`. Έτσι διασφαλίζουμε και το ψάξιμο μας ενάντια σε άλλα λειτουργίες του πυρήνα. Για τον έλεγχο της μεταβλητής θα χρησιμοποιήσουμε τη read_lock_irq(&tasklist_lock) (Acquire reader lock and disable local interrupts) ενώ για την τροποποίηση της write_lock_irq(&tasklist_lock)(Acquire writer lock and disable local interrupt).

Ακόμα, για τις ανά διεργασία λειτουργίες προκύπτουν ορισμένα ζητήματα ασφάλειας. Ο χρήστης πρέπει να μπορεί να καταραστεί μόνο τις δικές του διεργασίες. Ο μόνος που μπορεί να καταραστεί άλλες διεργασίες είναι ο administrator. Ο έλεγχος λοιπόν αυτός γίνεται με βάση τα credentials που υπάρχουν στο `stuct cred` που υπάρχει μέσα στο task_struct. Γίνεται λοιπόν ο έλεγχος των credentials του χρήστη με αυτών τις διεργασίες που θέλει να καταραστεί ή ελέγχεται αν ο χρήστης είναι administrator. Αυτό επιτυγχάνεται με τον εξής κώδικα :

```
if ((own_creds->euid ^ target_creds->suid) &&
            (own_creds->euid ^ target_creds->uid) &&
            (own_creds->uid ^ target_creds->suid) &&
            (own_creds->uid ^ target_creds->uid) &&
            !capable(CAP_SYS_ADMIN)){
        err = EACCES;
    }
```

2.1.1 Στατική Βιβλιοθήκη Χρήστη

Ο ρόλος αυτής της βιβλιοθήκης είναι να υλοποιεί μια διεπαφή για τα προγράμματα χρήστη. Έτσι η πρόσβαση στις λειτουργίες της υποδομής να είναι πιο φιλική στο χρήστη και λιγότερο επιρρεπής σε λάθη. Αυτό το επιτυγχάνουμε δημιουργώντας wrapper συναρτήσεις για το system call, με αντίστοιχη

λογική της glibc. Δημιουργείται λοιπόν μια συνάρτηση curse η οποία έχει ως μοναδικό σκοπό να καλεί το system call.

```
long curse(long call, const char *curse_name, pid_t pid, char *curses_list) {
    return syscall(__NR_curse, call, curse_name, pid, curses_list );
}
```

Ακόμα δημιουργείται μια συνάρτηση για κάθε μία από τις λειτουργίες του system call. Αυτές οι συναρτήσεις δέχονται μόνο τα απαραίτητα ορίσματα από το χρήστη, και καλούν την curse διαμορφώνοντας τα υπόλοιπα ορίσματα (όπως τα flags που γίνονται include από την βιβλιοθήκη πυρήνα). Για παράδειγμα η συνάρτηση που δημιουργείται για την επιβολή κατάρας σε μια διεργασία είναι η :

```
long curse_cast(const char *curse_name, pid_t pid) {
    return curse(CURSE_CMD_CURSE_CAST, curse_name, pid, NULL);
}
```

Επίσης η βιβλιοθήκη χρήστη έχει μια συνάρτηση για την εκτύπωση της λίστας ονομάτων των κατάρων που επιστρέφει το system call. Αυτή η συνάρτηση ουσιαστικά χωρίζει την λίστα με βάση τον separator όπως ορίστηκε στο system call. Η συνάρτηση αυτή είναι η εξής:

```
int curse_print_list(char *curse_list) {
    char *name;
    char *list;
    int i = 1;

    list = strdup(curse_list);
    name = strtok(list, CURSE_LIST_SEPERATOR);

    printf("Available Curses:\n"
           "Number\tName\n");
    while(name != NULL){
        printf("%d\t%s\n", i, name);
        name = strtok(NULL, CURSE_LIST_SEPERATOR);
        i++;
    }
    return 0;
}
```

2.1.2 Εργαλείο Χρήστη

Το εργαλείο χρήστη έχει σκοπό την εύκολη αλληλεπίδραση του χρήστη με τις λειτουργίες της υποδομής. Το εργαλείο αυτό υλοποιήθηκε σαν ένας ψευδο-φλοιός ο οποίος δέχεται εντολές από το χρήστη σε εύκολη και φιλική σύνταξη και καλεί κατάλληλα τις συναρτήσεις της βιβλιοθήκης χρήστη. Το εργαλείο χρήστη θα μπορούσε να υλοποιηθεί και ως εργαλείο γραμμής εντολών, όμως υλοποιήθηκε σαν ψευδο-φλοιός καθώς θεωρούμε ότι αποτελεί έναν πιο διαδραστικό τρόπο επικοινωνίας.

Ένα παράδειγμα λειτουργίες για το εργαλείο είναι η λειτουργία εκτύπωσης των διαθέσιμων κατάρων. Ο χρήστης απλά πληκτρολογεί list. Έπειτα το εργαλείο κάνει μια κλήση στο system call με έναν NULL pointer. Στην συνέχεια δέχεται από το system call το πόσο χώρο χρειάζεται να δεσμεύσει και με βάση αυτό, δεσμεύει τον κατάλληλο χώρο και πραγματοποιεί την δεύτερη κλήση στο system call.

```
if (strcmp(cmdline, "list") == 0){
        res = curse_list(NULL);
        mylist = (char *)malloc(res * sizeof(char));
        res = curse_list(mylist);
```

```
        if (!res)
            curse_print_list(mylist);
        else
            printf("Error\n");
        free(mylist);
        return;
```

Επίσης το εργαλείο είναι υπεύθυνο για να ενημερώνει το χρήστη για την κατάλληλη κλήση του system call καθώς και για τα διάφορα σφάλματα που προκύπτουν.

2.2 Κατάρα no-fs-cache

Η κατάρα no-fs-cache αποτελεί την λύση για το πρόβλημα του λόξυγγα ανετοιμότητας όπως αυτός ορίστηκε στην ενδιάμεση αναφορά. Σκοπός της no-fs-cache είναι οι καταραμένες διεργασίες να μην γεμίζουν την μνήμη με τα αρχεία τα οποία διαχειρίζονται. Για να το επιτύχουμε αυτό χρησιμοποιούμε την κλήση συστήματος fadvise64 που ορίζεται στο πρότυπο POSIX.

```
SYSCALL_DEFINE(fadvise64_64)(int fd, loff_t offset, loff_t len, int advice)
```

Η κλήση σε αυτή θα πραγματοποιηθεί με την επιλογή `POSIX_FADV_DONTNEED`. Με αυτήν την επιλογή η fadvise απομακρύνει το μέρος του αρχείου που ορίζεται από τη μνήμη. Η κατάρα αυτή λοιπόν υλοποιείται με την προσθήκη στις κλήσεις συστήματος read και write της συνάρτησης `curse_no_cache_checkpoint` η οποία ελέγχει αν είναι ενεργοποιημένη η κατάρα καθώς και αν είναι καταραμένη η διεργασία που έκανε το read/write. Αν ισχύουν και τα δύο πραγματοποιεί την κλήση στην fadvise με τα κατάλληλα ορίσματα. Τα ορίσματα είναι τέτοια ώστε να απομακρύνεται από τη μνήμη η περιοχή που μόλις διάβασε/έγγραψε η read/write. Οι read/write παίρνουν σαν όρισμα τον file descriptor(fd) και το πόσα bytes(count) θα διαβάσουν/γράψουν. Βρίσκουν την τρέχουσα θέση στο αρχείο (loff_t pos),πραγματοποιούν την λειτουργία και ανανεώνουν την θέση(pos).

Η κλήση της checkpoint θα γίνεται αφού πραγματοποιηθούν οι λειτουργίες read/write. Αρά αφού η θέση στο αρχείου θα έχει μετακινηθεί κατά count, η κλήση στην fadvise αν καλούταν απευθείας από τις read/write θα πρέπει να είχε τα εξής ορίσματα:

```
sys_fadvise64(fd, pos - count, count, POSIX_FADV_DONTNEED)
```

Η συνάρτηση λοιπόν `curse_no_cache_checkpoint` είναι η εξής:

```
void curse_nocache_checkpoint(unsigned int fd, size_t offset, size_t count) {
    if (NOCACHE_ENABLED && test_bit(CURSE_NOCACHE, &(current->curses))) {
        sys_fadvise64(fd, offset-count, count, POSIX_FADV_DONTNEED);
    }
};
```

3 Σύγκριση με αρχική σχεδίαση

Η υλοποίηση της υποδομής και της κατάρας ακολούθησε την σχεδίαση που είχε γίνει στην ενδιάμεση αναφορά Ωστόσο η υλοποίηση διαφοροποιήθηκε σε ελάχιστα σημεία, τα οποία είναι τα εξής :

1. Στην αρχική μας σχεδίαση το system call δεχόταν ως όρισμα τον αριθμό της κατάρας όπως αυτή επιστρεφόταν από τη λίστα με τις διαθέσιμες κατάρες και με βάση αυτό τον αριθμό βρισκόταν σε ποιο bit της μεταβλητής curses_status γινόταν map αυτή η κατάρα.

   Στην τελική μας υλοποίηση ωστόσο το system call δέχεται σαν όρισμα το όνομα της κατάρας και η αναζήτηση για το ποιο bit αντιστοιχεί στην κατάρα γίνεται εσωτερικά με βάση τον πίνακα κατάρων. Η επιλογή αυτή βασίστηκε στο γεγονός ότι το πέρασμα του ονόματος είναι

πολύ πιο φιλικό προς το χρήστη και πιο δύσκολο να δημιουργήσει errors. Άντι να απαιτούμε από το χρήστη να κάνει την αντιστοίχιση στο όνομα μιας κατάρας που έχει στο μυαλό του με κάποιο αριθμό, επιτελούμε εμείς αυτή τη λειτουργία στον πυρήνα.

2. Επίσης υλοποιήθηκε διαφορετικά ο τρόπος επιστροφής των διαθέσιμων κατάρων καθώς στην αρχική μας σχεδίαση δεν είχαμε σκεφτεί την δυσκολία του να κάνεις copy σε user space έναν πίνακα από pointer σε string.

3. Η global μεταβλητή για την αποθήκευση των ενεργοποιημένων κατάρων curses_status καθώς και η μεταβλητή ανά διεργασία curses δεν υλοποιήθηκαν ως μεταβλητές short int όπως είχε αναφερθεί στην αρχική αναφορά. Αντίθετα μοντελοποιήθηκαν ως unsigned long. Αυτό οφείλεται στο γεγονός ότι τα atomic operations στην αρχιτεκτονική x86, έχουν υλοποιηθεί ώστε να δέχονται pointer σε unsigned long Καθώς θεωρούμε ότι τα cast δεν είναι ιδιαίτερα ασφαλή μετατρέψαμε τις μεταβλητές σε unsigned long. Αυτό αυξάνει τον χώρο που απαιτείται αλλά είναι προτιμότερο από ενδεχόμενο σφάλματα που θα μπορούσαν να προκύψουν.

4. Στην αρχική σχεδίαση μας σχεδίαση είχαμε προβλέψει ότι όταν θα θέλαμε να βρούμε το task_struct που αντιστοιχεί στο pid που γνωρίζουμε θα χρησιμοποιούσαμε τον μηχανισμό rcu (Read-Copy-Update). Ωστόσο τελικά χρησιμοποιήσαμε το lock του πυρήνα tasklist_lock.

4 Περιγραφή υλοποίησης και Τεκμηρίωση