Βασική Διαχείρηση Μνήμης στο Linux Kernel
Λειτουργικά Συστήματα (ECE ΓΚ702)

University of Patras — 2022

1 Εισαγωγή

Στην εργασία αυτή θα γίνει εξοικείωση με μερικούς βασικούς μηχανισμούς δέσμευσης μνήμης και συγχρονισμού στο Linux kernel.

Το λειτουργικό σύστημα, μεταξύ άλλων, μπορεί να θεωρηθεί σαν μία διεπαφή υψηλού επιπέδου για το υλικό το οποίο τρέχει στο εκάστοτε σύστημα.

2 Το Linux Kernel API

Ο προγραμματισμός modules για το Linux Kernel έχει αρκετές διαφορές από τη συγγραφή κλασικών προγραμμάτων σε γλώσσα C. Βασική διαφορά είναι η έλλειψη της βασικής βιβλιοθήκης (libc). Ένας προγραμματιστής θα πρέπει να χρησιμοποιεί τις συναρτήσεις που του γίνονται διαθέσιμες από το Linux kernel για πράξεις όπως δέσμευση μνήμης και συγχρονισμό. Προφανώς, υπάρχει η δυνατότητα προσθήκης νέων εργαλείων και συναρτήσεων μέσα στον πυρήνα.

Όταν αναπτύσσουμε κανονικές εφαρμογές, ο κώδικάς μας έχει χαμηλές δικαιοδοσίες. Οι πληροφορίες και οι πόροι στα οποία έχει πρόσβαση πάντα διαχειρίζονται από το λειτουργικό σύστημα. Αυτό έχει σαν αποτέλεσμα την υψηλότερη ασφάλεια αλλά και τη διευκόλυνση του προγραμματιστή αφού σύνθετοι μηχανισμοί όπως η δέσμευση μνήμης "κρύβονται" πίσω από συστήματα όπως η εικονική μνήμη.

2.1 Δέσμευση Μνήμης

Σε εφαρμογές που συνήθως αναπτύσσουμε η δέσμευση μνήμης γίνεται μέσω της συνάρτησης `malloc`. Ωστόσο, αναπτύσσοντας κώδικα σε περιβάλλον kernel έχουμε στη διάθεσή μας διαφορετικά είδη μνήμης:

- Φυσική Μνήμη
- Εικονική Μνήμη από το χώρο διευθύνσεων του πυρήνα
- Εικονική Μνήμη από το χώρο διευθύνσεων μιας διεργασίας
- Resident Μνήμη - όταν ξέρουμε με βεβαιότητα οι σελίδες μνήμης είναι παρούσες στη φυσική μνήμη

Ενδεικτικά, το πού θα συναντήσετε το κάθε είδος μνήμης:

- Τα δεδομένα ενός module βρίσκονται πάντα σε resident μνήμη.
- Μνήμη στο χώρο διευθύνσεων μιας διεργασίας δεν είμαστε σίγουροι πως είναι παρούσα στη φυσική μνήμη λόγω του μηχανισμού paging του λειτουργικού συστήματος.
- Μνήμη στο χώρο διευθύνσεων του πυρήνα μπορεί να είναι resident ή όχι.

- Δυναμική μνήμη μπορεί να είναι resident ή όχι με βάση το πώς δεσμεύτηκε.

Όταν χειριζόμαστε resident μνήμη, δηλαδή μνήμη που πράγματι υπάρχει στη φυσική μνήμη του συστήματός μας, μπορούμε απλά να έχουμε πρόσβαση σε οποιαδήποτε θέση της. Ωστόσο, non-resident μνήμη μπορεί να εκτελεστεί μόνο σε πλαίσιο διεργασίας (process context). Ο πυρήνας βρίσκεται σε process context όταν για παράδειγμα καλέσετε μία κλήση συστήματος (πχ. `fork`) μέσα από ένα πρόγραμμά σας.

Για να δεσμεύσουμε resident μνήμη μέσα από το kernel χρησιμοποιούμε τη συνάρτηση `kmalloc`.

Σημείωση Η μνήμη που δεσμεύεται από την `kmalloc` είναι συνεχής στη φυσική μνήμη. Αυτό είναι σε αντίθεση από την κλασσική συνάρτηση `malloc` η οποία μας επιστρέφει μνήμη η οποία δεν είναι συνεχής αλλά είναι φαινομενικά συνεχής μέσω του μηχανισμού εικονικής μνήμης του λειτουργικού συστήματος.

kmalloc-example.c

```c
#include <linux/slab.h>

// ...

// allocate 100 bytes of continuous memory
char* my_memory = kmalloc(100 * sizeof(char),
        GFP_KERNEL);

// ...
```

Η χρήση της είναι πολύ παρόμοια με τη γνωστή συνάρτηση `malloc` της βασικής βιβλιοθήκης της c. Το πρώτο όρισμο είναι το μέγεθος της δεσμευμένης περιοχής μνήμης ενώ το δεύτερο όρισμα δείχνει τον τρόπο με τον οποίο πρέπει να γίνει αυτή η δέσμευση.

Μερικά από τα flags που μπορούν να χρησιμοποιηθούν για τη δέσμευση μνήμης είναι

- `GFP_KERNEL` όπου δεσμεύεται κανονικά μνήμη πυρήνα. Ενδεχομένως αν καλεστεί η `kmalloc` με αυτό το flag από μία διεργασία, η διεργασία αυτή μπορεί να μπει σε αναμονή αν το σύστημα βρίσκεται σε κατάσταση χαμηλής μνήμης. Ο πυρήνας θα λάβει τις κατάλληλες ενέργειες για να δεσμεύσει την απαιτούμενη μνήμη.
- `GFP_ATOMIC` όπου δεσμεύεται μνήμη πυρήνα σε μέρη όπως interrupt handlers ή kernel timers. Εδώ, η τρέχουσα διεργασία δεν γίνεται να διακοπεί. Έτσι, ο πυρήνας δεσμεύει άμεσα ακόμα και την τελευταία ελεύθερη σελίδα (page).
- `__GFP_DMA` όπου ζητείται να δεσμευτεί μνήμη η οποία μπορεί να χρησιμοποιηθεί σε DMA (Direct Memory Access) από και προς συσκευές. Το ακριβές της νόημα εξαρτάται από την πλατφόρμα στην οποία βρισκόμαστε.

Σε όλα αυτα τα flags το πρόθεμα `GFP` αναφέρεται στη συνάρτηση `get_free_pages` η οποία καλείται εσωτερικά από την `kmalloc`.

2.2 Μηχανισμοί Συγχρονισμού

2.2.1 Spinlock

Ο μηχανισμός spinlock είναι ένας από τις πιο απλές μεθόδους συγχρονισμού. Βεβαιώνει πως ο κώδικας μέσα στο spinlock block εκτελείται μόνο από ένα νήμα. Ωστόσο, αποκλεισμός μέσω spinlocks είναι ακατάλληλος για critical regions με μεγάλο χρόνο εκτέλεσης διότι άλλοι threads που θέλουν να έχουν πρόσβαση στο critical region θα κάνουν συνεχή έλεγχο της διαθεσιμότητας (θα κάνουν spin).

atomic_t-example.c

```c
#include <linux/spinlock.h>

DEFINE_SPINLOCK(lock1);
// (same as) spin_lock_init(&lock1)

spin_lock(&lock1);
/* critical region */
spin_unlock(&lock1);
```

Περισσότερες πληροφορίες για τις λεπτομέρειες χρήσης των spinlocks μπορείτε να βρείτε στο kernel documentation.

2.2.2 Mutex

Τα mutexes αναπαρίστανται στο Linux kernel ως δομές `struct mutex`. Η χρήση τους και η λειτουργία τους είναι παρόμοια με αυτή των κλασσικών mutexes σε user space. Το mutex αποκτάται (lock) πριν το critical region και απελευθερώνεται (unlock) στο τέλος της.

2.2.3 Ατομικές Μεταβλητές (Atomic Variables)

Για τον συγχρονισμό της πρόσβασης σε μία μεταβλητή, το Linux kernel προσφέρει ατομικές ματαβλητές μέσω του τύπου `atomic_t` ο οποίος κρατάει μια ακέραια τιμή. Αυτό είναι μια διεπαφή για την χρήση των RMW (Read Modify Write) λειτουργιών των διάφορων επεξεργαστών. Αυτές οι ατομικές λειτουργίες μπορούν να διαβάσουν μία θέση μνήμης και να γράψουν σε αυτή ταυτόχρονα.

atomic_t-example.c

```c
#include <asm/atomic.h>

void atomic_set(atomic_t *v, int i);
int atomic_read(atomic_t *v);
void atomic_add(int i, atomic_t *v);
void atomic_sub(int i, atomic_t *v);
void atomic_inc(atomic_t *v);
void atomic_dec(atomic_t *v);
int atomic_inc_and_test(atomic_t *v);
int atomic_dec_and_test(atomic_t *v);
int atomic_cmpxchg(atomic_t *v, int old, int new);
```

Σημείωση Ο τύπος `atomic_t` ορίζεται στο αρχείο `linux/include/linux/atomic.h`. Στην ουσία είναι ένα περίβλημα γύρω από γνωστούς τύπους ακεραίων (`int`, `long int`).

Η χρήση των ατομικών μεταβλητών μπορεί να γίνει για την αποκλειστική πρόσβαση σε έναν πόρο του συστήματος όπως μία συσκευή.

3 Ασκήσεις

Άσκηση 1

Βρείτε στον πηγαίο κώδικα του Kernel τους ορισμούς των συμβόλων. Μπορείτε να χρησιμοποιήσετε το εργαλείο αναζήτησης LXR

1. `kmalloc`
2. `kfree`
3. `get_free_pages`
4. `atomic_t`
5. `atomic_read`

Εξηγήστε τη σημασία τους.

Άσκηση 2

Γράψτε ένα Kernel module το οποίο όταν φορτωθεί θα δεσμεύει μνήμη μεγέθους 4096 bytes και στη συνέχεια θα τυπώνει τα περιεχόμενά της. Φροντίστε με την εκφόρτωση του module να απελευθερώνεται η μνήμη.

Άσκηση 3

Φτιάξτε ένα kernel module το οποίο θα βρίσκει το `task_struct *` που θα αντιστοιχεί σε μία διεργασία με ένα δοθέντο `PID` και εξετάστε το `mm` μέλος του `task_struct *`. Τυπώστε το πεδίο του `mm_users`. Αυτό περιέχει τον αριθμό των διεργασιών που μοιράζονται αυτή τη θέση μνήμης. Βασιστείτε στον κώδικα που δίνεται στο αρχείο `process-mm-module.c`.

Ξεκινήστε μία διεργασία η οποία θα δημιουργεί μερικά threads. Μπορείτε να βασιστείτε στον παρακάτω κώδικα.

threads.c

```c
#include <unistd.h>
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>

#define THREADS 4

void*
thread_func(void * __args)
{
    sleep(1);
    return NULL;
}

int main()
{
    printf("PID: %d\n", getpid());

    // sleep(5);

    pthread_t my_threads[THREADS];

    for (int i = 0; i < THREADS; i++)
        pthread_create(&my_threads[i],
                NULL, thread_func, NULL);

    for (int i = 0; i < THREADS; i++)
        pthread_join(my_threads[i], NULL);
}
```

Τρέξτε τον παραπάνω κώδικα χρησιμοποιώντας τις εντολές:

Command Line

```
$ gcc -o thread threads.c
$ ./threads
```

Μόλις αρχίσει να τρέχει, η διεργασία αυτή θα τυπώσει το PID της. Αν έχετε ήδη φορτώσει το module σας, ξεφορτώστε το και επαναφορτώσετε το ορίζοντας σε αυτό το PID του προγράμματος `threads`.

Στον κώδικα έχει προστεθεί ένα kernel parameter ώστε να γίνει πιο εύκολος ο ορισμός του νέου PID. Τα kernel parameters είναι χρήσιμα για την παραμετρικοποίηση ενός module κατά τη φόρτωση. Πλέον, όταν φορτώνετε το module, μπορείτε να ορίσετε διαφορετικό PID διεργασίας το οποίο θα ψάξει χωρίς να χρειάζεται να το επαναμεταγλωττίσετε.

Ωστόσο, αν κάνετε αλλαγές στον κώδικα του module θα πρέπει όπως πριν να το μεταγλωττίσετε ξανά.

Command Line

```
$ sudo insmod process-mm-module PID=$PID # specify PID when loading module
$ sudo rmmod process-mm-module # unload normally
```

Τί παρατηρείτε όταν το module σας τυπώνει την τιμή του πεδίου `task->mm->mm_users`?

4 Πηγές

- Linux Kernel Module Programming Guide
- Linux Device Drivers
- Operating System Concepts 9th Edition - Silbershatz, Galvin, Gagne (ch. 18)