



УНИВЕРЗИТЕТ
У НОВОМ САДУ



ФАКУЛТЕТ
ТЕХНИЧКИХ НАУКА

Трг Доситеја Обрадовића 6, 21000 Нови Сад, Југославија
Деканат: 021 350-413; 021 450-810; Централа: 021 350-122
Рачуноводство: 021 58-220; Студентска служба: 021 350-763
Телефакс: 021 58-133; e-mail: ftndean@uns.ns.ac.yu



Сертификован
систем
квалитета



PROJEKAT

Iz Mikroračunarskih sistema za rad u realnom vremenu

TEMA PROJEKTA:

Surf drajver koji simulira rad sa IP jezgrom napravljenim u VHDL programskom jeziku.

TEKST ZADATKA:

Napisati drajver koji radi sa aplikacijom iz specifikacije.

Studenti:

Ristić Dejana 56/2019

Vig Aleksandar 142/2019

Žarković Nemanja 69/2019

Mentor:

Nebojša Pilipović

U Novom Sadu, 30.09.2024.

1.Opis projekta

U našem projektu ideja je bila da se napravi drajver koji će modifikovati funkciju **createVector**, koja je na osnovu odrađenog profajliranja funkcija koja oduzima najviše sistemskog vremena, tako da se ona može pokrenuti u C programskom jeziku i raditi kao kernel modul.

Potrebno je bilo da se u drajver upišu vrednosti koje se ne nalaze u našoj funkciji, i zatim da se one primene u funkciji. Pošto funkcija radi sa adresama, napravljena su dva nova niza koja će skladištiti adrese nizova iz aplikacije, i raditi sa njima u drajveru(**pixels1D i lookup2**), a u **_index1D** nizu su skladištene vrednosti koje se dobijaju na izlazu iz funkcije i šalju nazad aplikaciji, na dalju obradu koda.

Funkcija **createVector** obrađuje uzorak slike na osnovu parametara **scale, row i col**. U njoj se vrši proračun kako bi se odredile pozicije i vrednosti piksela u slici, koje se koriste za ažuriranje vektora **_index1D** sa izračunatim vrednostima. Funkcija koristi dve ugnježdene for petlje da bi prošla kroz sve piksele unutar kvadrata određenog radijusom **iradius**. Svaki piksel prolazi kroz transformaciju koordinata da bi se prilagodio orijentaciji ugla slike. Proverava se da li piksel pripadaju slici, i ako da izračunavaju se gradijenti slike, a zatim se računaju horizontalni **dx** i vertikalni **dy** gradijenti korišćenjem sinusa i kosinusa.

Na osnovu gradijenata, funkcija računa težine **rweight1, rweight2, cweight1 i cweight2**, koje se koriste za računanje vrednosti uzorka slike.

Na kraju ako pikseli pripadaju slici, funkcija ažurira **_index1D**, koji sadrži rezultate za svaku orijentaciju **ori1, ori2** i koordinate slike **ri, ci**

```

void createVector(double scale, double row, double col) {
    int iradius, iy, ix;
    double spacing, radius, rpos, cpos, rx, cx;
    int step = max((int)(scale/2 + 0.5), 1);
    iy = (int) (row + 0.5);
    ix = (int) (col + 0.5);
    double fracy = row - iy;
    double frax = col - ix;
    double fracr = COSE * fracy + SINE * frax;
    double fracc = - SINE * fracy + COSE * frax;
    spacing = scale * _MagFactor;
    radius = 1.4 * spacing * (_IndexSize + 1) / 2.0;
    iradius = (int) (radius/step + 0.5);

    for (int i = 0; i <= 2*iradius; i++) {
        for (int j = 0; j <= 2*iradius; j++) {
            double temp1_rpos, temp1_rpos, temp1_rpos, temp1_rpos;
            double temp1_cpos, temp1_cpos, temp1_cpos, temp1_cpos;

            temp1_rpos = COSE * (i - iradius);
            temp1_rpos = SINE * (j - iradius);
            temp1_rpos = temp1_rpos + temp1_rpos;
            temp1_rpos = step * temp1_rpos - fracr;
            rpos = temp1_rpos / spacing;

            temp1_cpos = - SINE * (i - iradius);
            temp1_cpos = COSE * (j - iradius);
            temp1_cpos = temp1_cpos + temp1_cpos;
            temp1_cpos = step * temp1_cpos - fracc;
            cpos = temp1_cpos / spacing;

            rx = rpos + 2.0 - 0.5;
            cx = cpos + 2.0 - 0.5;

            if (rx > -1.0 && rx < (double) _IndexSize &&
                cx > -1.0 && cx < (double) _IndexSize) {

                int r = iy + (i - iradius) * step;
                int c = ix + (j - iradius) * step;
                int ori1, ori2;
                int ri, ci;

                int addSampleStep = (int)scale;
                double weight;
                double dx1, dx2, dy1, dy2;
                double dx, dy;
                double dxw, dyw;
                double rfrac, cfrac;
                double rweight1, rweight2, cweight1, cweight2;
                double dx1, dx2, dy1, dy2;

                if (r >= 1 + addSampleStep && r < height - 1 - addSampleStep && c >= 1 + addSampleStep && c < width - 1 - addSampleStep) {
                    weight = _lookup2((int)(rpos * rpos + cpos * cpos));

                    dx1 = pixelsID[(r + addSampleStep + 1) * width + (c + addSampleStep + 1)]
                        + pixelsID[(r - addSampleStep) * width + c]
                        - pixelsID[(r - addSampleStep) * width + (c + addSampleStep + 1)]
                        - pixelsID[(r + addSampleStep + 1) * width + c];

                    dx2 = pixelsID[(r + addSampleStep + 1) * width + (c + 1)]
                        + pixelsID[(r - addSampleStep) * width + (c - addSampleStep)]
                        - pixelsID[(r - addSampleStep) * width + (c + 1)]
                        - pixelsID[(r + addSampleStep + 1) * width + (c - addSampleStep)];

                    dy1 = pixelsID[(r + 1) * width + (c + addSampleStep + 1)]
                        + pixelsID[(r - addSampleStep) * width + (c - addSampleStep)]
                        - pixelsID[(r - addSampleStep) * width + (c + addSampleStep + 1)]
                        - pixelsID[(r + 1) * width + (c - addSampleStep)];

                    dy2 = pixelsID[(r + addSampleStep + 1) * width + (c + addSampleStep + 1)]
                        + pixelsID[r * width + (c - addSampleStep)]
                        - pixelsID[r * width + (c + addSampleStep + 1)]
                        - pixelsID[(r + addSampleStep + 1) * width + (c - addSampleStep)];

                    dxw = weight * (dx1 - dx2);
                    dyw = weight * (dy1 - dy2);
                    dx1 = COSE * dxw;
                    dx2 = SINE * dyw;
                    dx = dx1 + dx2;
                    dy1 = SINE * dxw;
                    dy2 = COSE * dyw;
                    dy = dy1 - dy2;
                    if (dx < 0) ori1 = 0;
                    else ori1 = 1;

                    if (dy < 0) ori2 = 2;
                    else ori2 = 3;

                    if (rx < 0) ri = 0;
                    else if (rx >= _IndexSize) ri = _IndexSize - 1;
                    else ri = rx;

                    if (cx < 0) ci = 0;
                    else if (cx >= _IndexSize) ci = _IndexSize - 1;
                    else ci = cx;

                    rfrac = rx - ri;
                    cfrac = cx - ci;

                    if (rfrac < 0.0) rfrac = 0.0;
                    else if (rfrac > 1.0) rfrac = 1.0;

                    if (cfrac < 0.0) cfrac = 0.0;
                    else if (cfrac > 1.0) cfrac = 1.0;

                    rweight1 = dx * (1.0 - rfrac);
                    rweight2 = dy * (1.0 - rfrac);
                    cweight1 = rweight1 * (1.0 - cfrac);
                    cweight2 = rweight2 * (1.0 - cfrac);

                    if (ri >= 0 && ri < _IndexSize && ci >= 0 && ci < _IndexSize) {
                        _indexID[ri * (_IndexSize * _OriSize) + ci * _OriSize + ori1] = cweight1;
                        _indexID[ri * (_IndexSize * _OriSize) + ci * _OriSize + ori2] = cweight2;
                    }
                }
            }
        }
    }
}

```

Slika 1-1/2 Izgled funkcije createVector u drajveru

2.Drajver

U ovom poglavlju opisaće se rad funkcija koje su potrebne da bi drajver funkcionisao.

Prvo je napravljena struktura **file_operations**, koja definiše skup funkcija koje kernel koristi za interakciju sa fajlovima u okviru drajvera. Te funkcije su osnovni mehanizmi za rad sa uređajima preko drajvera, dok kernel osigurava da je modul pravilno referenciran i ne može biti uklonjen dok god postoji aktivna interakcija sa njim.

1. .owner =THIS_MODULE

Obezbeđuje da kenel zna kojem modulu ove operacije pripadaju. Makro pokazuje na trenutni modul, odnosno drajver koji koristi ovu strukturu. To pomaže kernelu da održava referencu na broj učitavanja modula, sprečavajući njegovo uklanjanje dok se funkcije koriste.

2. .open = surf_driver_open,

Funkcija se poziva svaki put kada korisnički proces otvori uređaj povezan sa drajverom

3. .release = surf_driver_release

Ova funkcija se poziva svaki put kada korisnički proces zatvori uređaj (pozove close sistemski poziv). Koristi se za pravilno zatvaranje drajvera i uređaja, i oslobađa memoriju

4. .read = surf_driver_read

Ova funkcija se koristi za čitanje podataka iz drajvera u korisnički prostor.

Omogućava korisničkim aplikacijama da čitaju podatke iz drajvera

5. .write = surf_driver_write

Ova funkcija omogućava korisničkom prostoru da piše podatke na uređaj.Koristi se kada korisnički proces pošalje podatke drajveru, a on će rukovati načinom kako i gde podaci idu.

```
static const struct file_operations surf_driver_fops = {
    .owner = THIS_MODULE,
    .open = surf_driver_open,
    .release = surf_driver_release,
    .read = surf_driver_read,
    .write = surf_driver_write,
};
```

Slika 2 file_operations stuktura

Nakon ove stukture slede funkcije open i release koje se pozivaju kada kernel otvori i zatvori uređaj. Njihova povratna vrednost je 0 što znači da je otvaranje odnosno zatvaranje uređaja bilo uspešno.

```
int surf_driver_open(struct inode *inode, struct file *file) {
    return 0;
}

int surf_driver_release(struct inode *inode, struct file *file) {
    return 0;
}
```

Slika 3 surf_driver_open i surf_driver_release funkcije

Funkcija `init_module()` inicijalizuje kernel modul u Linux kernelu. Poziva se kada se modul učitava u kernel pomoću alata **insmod**. Funkcija koristi *printk* za ispis informacija u kernel logove (dostupno putem komande *dmesg*).

Preko funkcije **register_chrdev()** registruje se karakterni uređaj. Ovaj sistemski poziv registruje uređaj u kernelu i vraća **major number** (glavni broj uređaja), koji identifikuje uređaj.

Kernel pokušava da alocira memoriju za niz piksela `pixels1D`, koristeći `kmalloc()`. Flag `GFP_KERNEL` označava da je memorija alocirana u kontekstu kernela.

Ako alokacija ne uspe, funkcija vraća grešku `-ENOMEM` i oslobađa resurse registracije uređaja pomoću `unregister_chrdev()`. Isto se radi za `index1D` niz samo je drugačija alokacija memorije.

Ako su sve operacije uspešno izvršene, funkcija ispisuje poruku da je uređaj uspešno registrovan sa odgovarajućim major brojem i vraća se nula kao uspešna inicijalizacija modula.

```
int init_module(void) {
    printk(KERN_INFO "\n");
    printk(KERN_INFO "surf driver starting insmod.\n");
    /*if (mutex_lock_interruptible(&surf_mutex)) {
        printk(KERN_ALERT "Nije moguće zaključati mutex\n");
        return -EINTR; // Vraća grešku ako je došlo do prekida
    }*/
    // Registracija uređaja
    major_num = register_chrdev(0, DEVICE_NAME, &surf_driver_fops);
    if (major_num < 0) {
        printk(KERN_ALERT "Failed to register a major number\n");
        return major_num;
    }

    // Alokacija memorije za pixels1D (npr. 1D niz piksela)
    pixels1D = kmalloc(width * height * sizeof(double), GFP_KERNEL);
    if (!pixels1D) {
        printk(KERN_ALERT "Failed to allocate memory for pixels1D\n");
        unregister_chrdev(major_num, DEVICE_NAME);
        return -ENOMEM;
    }

    // Alokacija memorije za _index1D (ako je potrebna)
    _index1D = kmalloc(_IndexSize * _OriSize * sizeof(double), GFP_KERNEL);
    if (!_index1D) {
        printk(KERN_ALERT "Failed to allocate memory for _index1D\n");
        kfree(pixels1D); // Oslobodi prethodno alociranu memoriju
        unregister_chrdev(major_num, DEVICE_NAME);
        return -ENOMEM;
    }

    printk(KERN_INFO "Registered correctly with major number %d\n", major_num);
    //mutex_unlock(&surf_mutex);
    return 0; // Uspešna inicijalizacija
}
```

Slika 4 init_module

Funkcija **cleanup_module()** je funkcija za deinicijalizaciju kernel modula, koja se poziva kada se modul uklanja iz kernela preko komande `rmmod`. Ova funkcija je zadužena za oslobađanje svih resursa koje je modul alocirao tokom svog rada, poput memorije i registracija uređaja.

U njoj se oslobađa memorija za sva tri niza `pixels1D`, `lookup2` i `_index1D` tako što se za svaki proverava da li je alociran, ako jeste memorija se oslobađa preko `kfree()`, a pokazivač se stavlja na `NULL` kako bi se sprečilo dalje korišćenje nevalidnog pokazivača.

Na kraju se major broj preko `unregister_chrdev()` odregistruje i ispisuje se poruka da je oslobodjen drajver.

```
void cleanup_module(void) {
    /* Oslobađanje pixels1D ako je alociran */
    if (pixels1D != NULL) {
        kfree(pixels1D);
        printk(KERN_INFO "pixels1D je oslobođen\n");
    }

    /* Oslobađanje _lookup2 ako je alociran */
    if (_lookup2 != NULL) {
        kfree(_lookup2); // Oslobađanje memorije za _lookup2
        _lookup2 = NULL; // Postavljanje pokazivača na NULL da spreči dalje korišćenje
        printk(KERN_INFO "_lookup2 je oslobođen\n");
    } else {
        printk(KERN_INFO "_lookup2 nije alociran ili je već oslobođen\n");
    }

    /* Oslobađanje ptInsurf ako je alociran */
    if (ptInsurf)
        kfree(ptInsurf);

    // Oslobađanje surfCenters ako je alociran
    if (surfCenters)
        kfree(surfCenters);

    // Provera da li je niz _index1D alociran, ako jeste, oslobađamo ga
    if (_index1D != NULL) {
        kfree(_index1D); // Oslobađanje memorije za _index1D
        _index1D = NULL; // Postavljanje pokazivača na NULL da spreči dalje korišćenje
        printk(KERN_INFO "_index1D je oslobođen\n");
    } else {
        printk(KERN_INFO "_index1D nije alociran ili je već oslobođen\n");
    }

    // Unregistering the device
    unregister_chrdev(major_num, DEVICE_NAME);
    printk(KERN_INFO "Unregistered %s device\n", DEVICE_NAME);
}
```

Slika 5 `cleanup_module` funkcija

Sada se poziva funkcija `surf_driver_write` koja implementira osnovnu logiku za primanje i obradu podataka, uključujući parametre slike i nizova podataka(`pixels1D` i `_lookup2`).

Deklaracija funkcije:

- `file`: Struktura koja sadrži informacije o otvorenom fajlu
- `user_buff`: Bafer iz korisničkog prostora, koji sadrži podatke koje korisnička aplikacija šalje drajveru
- `size`: Veličina podataka koja se prenosi
- `offset`: Pozicija unutar fajla sa koje se piše

Prvo se proverava da li veličina bafera size odgovara očekivanoj veličini podataka. U slučaju greške vraća se greška –EINVAL koja predstavlja Invalid Argument.

Nakon toga preko funkcije **copy_from_user** preuzimaju se vrednosti iz korisničkog prostora, a ako se pojavi greška vraća se –EFAULT, segmentation fault.

Zatim se prvo oslobađa prethodno alocirana memorija pixels1D niza, a onda dolazi do njegove alokacije i kopiranja podataka iz korisničkog bafera u ovaj niz. Isto se radi i za lookup2 niz.

Kopirane vrednosti se onda čuvaju u odgovarajućim kernel promenljivama koje se dalje koriste za obradu u drajveru.

Imamo promenljive koje će izračunati skalirane vrednosti na osnovu ulaznih podataka i poziv funkcije createVector koja vrši dalju obradu na osnovu ovih vrednosti.

Na kraju funkcija vraća broj bajtova koji je uspešno napisan, što odgovara veličini bafera.

```
ssize_t surf_driver_write(struct file *file, const char __user *user_buff, size_t size, loff_t *offset) {
    double scale, row, col;
    double sine, cose;

    // Zaključaj mutex pre pristupa kritičnim resursima
    /*if (mutex_lock_interruptible(&surf_mutex)) {
        printk(KERN_ALERT "Nije moguće zaključati mutex\n");
        return -EINTR; // Vraća grešku ako je došlo do prekida
    }*/
    // Očekivana veličina podataka: scale, row, col, SINE, COSE + pixels1D + lookup2
    size_t expected_size = sizeof(double) * 5 + width * height * sizeof(double) + 40 * sizeof(double);

    // Proveri da li je ukupna veličina bafera ispravna
    if (size != expected_size) {
        printk(KERN_ALERT "Neispravna veličina bafera, očekivano %zu, primljeno %zu\n", expected_size, size);
        return -EINVAL;
    }

    // Kopiraj osnovne parametre: scale, row, col, SINE, COSE
    if (copy_from_user(&scale, user_buff, sizeof(double))) {
        printk(KERN_ALERT "Failed to copy image scale from user space\n");
        return -EFAULT;
    }

    if (copy_from_user(&row, user_buff + sizeof(double), sizeof(double))) {
        printk(KERN_ALERT "Failed to copy image rows from user space\n");
        return -EFAULT;
    }

    if (copy_from_user(&col, user_buff + 2 * sizeof(double), sizeof(double))) {
        printk(KERN_ALERT "Failed to copy image columns from user space\n");
        return -EFAULT;
    }

    if (copy_from_user(&sine, user_buff + 3 * sizeof(double), sizeof(double))) {
        printk(KERN_ALERT "Failed to copy image sine from user space\n");
        return -EFAULT;
    }

    if (copy_from_user(&cose, user_buff + 4 * sizeof(double), sizeof(double))) {
        printk(KERN_ALERT "Failed to copy image cose from user space\n");
        return -EFAULT;
    }

    // Oslobodi prethodno alociran 'pixels1D' ako postoji
    if (pixels1D != NULL) {
        kfree(pixels1D);
        printk(KERN_INFO "Prethodni pixels1D je osloboden\n");
    }

    // Alociraj memoriju za novi 'pixels1D'
    size_t pixels_size = width * height * sizeof(double);
    pixels1D = kmalloc(pixels_size, GFP_KERNEL);
    if (!pixels1D) {
        printk(KERN_ALERT "Neuspešna alokacija memorije za pixels1D\n");
        return -ENOMEM;
    }

    // Kopiraj niz 'pixels1D' iz korisničkog prostora
    if (copy_from_user(pixels1D, user_buff + 5 * sizeof(double), pixels_size)) {
        printk(KERN_ALERT "Greška prilikom kopiranja niza pixels1D iz korisničkog prostora\n");
        kfree(pixels1D); // Oslobodi memoriju u slučaju greške
        return -EFAULT;
    }

    printk(KERN_INFO "pixels1D uspešno kopiran iz aplikacije u kernel\n");

    // Kopiraj niz 'lookup2'
    size_t lookup2_offset = 5 * sizeof(double) + pixels_size;
    if (copy_from_user(_lookup2, user_buff + lookup2_offset, 40 * sizeof(double))) {
        printk(KERN_ALERT "Greška prilikom kopiranja niza _lookup2 iz korisničkog prostora\n");
        return -EFAULT;
    }

    printk(KERN_INFO "Niz _lookup2 uspešno inicijalizovan u kernelu\n");

    // Čuvanje vrednosti u kernel promenljivama
    data.ipoint_scale = scale;
    data.ipoint_row = row;
    data.ipoint_col = col;
    SINE = sine;
    COSE = cose;

    // Izračunavanje svake promenljive posebno
    double scaledValue = 1.65 * (1 + _doubleImage) * data.ipoint_scale;
    double colValue = (1 + _doubleImage) * data.ipoint_col;
    double rowValue = (1 + _doubleImage) * data.ipoint_row;

    // Pozivanje funkcije sa prethodno izračunatim vrednostima
    createVector(scaledValue, colValue, rowValue);

    printk(KERN_INFO "Received data: scale=%lf, row=%lf, col=%lf, sine=%lf, cose=%lf\n", scale, row, col, sine, cose);
    //mutex_unlock(&surf_mutex);
    return size; // Vraća uspešnu veličinu bafera
}
```

Slika 6-1/2 surf_driver_write funkcija

Funkcija surf_driver_read implementira operaciju čitanja podataka iz drajvera u korisnički prostor.

1. **struct file *file** – Pokazivač na strukturu file koja sadrži informacije o otvorenom fajlu. Ova struktura omogućava kernelu da prati status fajla koji je otvoren.
2. **char __user *user_buf** – Pokazivač na bafer u korisničkom prostoru u koji će se kopirati podaci iz kernela (drajvera).

- **size_t count** – Maksimalan broj bajtova koje aplikacija zahteva da pročita iz drajvera.
- **loff_t *ppos** – Pokazivač na offset (poziciju) unutar fajla ili uređaja, koji pokazuje trenutnu poziciju čitanja. Ova vrednost se koristi za tačno čitanje podataka, omogućavajući čitanje od određene pozicije u fajlu.

Računamo ukupnu veličinu niza `_index1D` preko dimenzija `_IndexSize` i `_OriSize`.

Funkcija `prekoremaining_bytes` proverava koliko je bajtova ostalo za čitanje, tj. razliku između ukupne veličine niza i trenutne pozicije čitanja.

Ako je ta razlika 0, to znači da su svi podaci već pročitani i vraća se 0, što označava kraj fajla (EOF).

Funkcija preko `bytes_to_read` određuje koliko bajtova može da pročita. Ako je zahtevano više bajtova nego što je ostalo za čitanje, pročitace se samo preostali bajtovi.

Zatim se kopiraju podaci iz kernela u korisnički prostor preko funkcije `copy_to_user()`.

Nakon toga se ažurira pozicija za čitanje preko `ppos(offset)` za broj bajtova koji su upravo pročitani.

I na kraju funkcija ispisuje koliko bajtova je pročitano i vraća broj bajtova koji su uspešno prebačeni u korisnički prostor.

```
ssize_t surf_driver_read(struct file *file, char __user *user_buf, size_t count, loff_t *ppos) {
    int IndexSize = _IndexSize;
    int OriSize = _OriSize;
    size_t total_size = IndexSize * IndexSize * OriSize * sizeof(double); // Ukupna veličina _index1D niza
    size_t bytes_to_read;
    size_t remaining_bytes;

    /* if (mutex_lock_interruptible(&surf_mutex)) {
        printk(KERN_ALERT "Nije moguće zaključati mutex\n");
        return -EINTR; // Vraća grešku ako je došlo do prekida
    }*/

    printk(KERN_INFO "surf_driver: Read request, count = %zu, ppos = %lld\n", count, *ppos);

    // Provera koliko bajtova je preostalo za čitanje
    remaining_bytes = total_size - *ppos;
    if (remaining_bytes == 0) {
        printk(KERN_INFO "surf_driver: No more data to read (EOF)\n");
        return 0; // Kraj fajla
    }

    // Odredi koliko bajtova treba pročitati
    bytes_to_read = min(count, remaining_bytes); // uzimamo najmanju vrednost između tražene količine i preostalih bajtova

    // Kopiraj podatke iz _index1D niza u korisnički prostor
    if (copy_to_user(user_buf, _index1D + (*ppos / sizeof(double)), bytes_to_read)) {
        printk(KERN_ALERT "surf_driver: Failed to copy _index1D to user space\n");
        return -EFAULT;
    }

    // Ažuriraj ppos
    *ppos += bytes_to_read;

    printk(KERN_INFO "surf_driver: Copied %zu bytes to user space, new ppos = %lld\n", bytes_to_read, *ppos);
    //mutex_unlock(&surf_mutex);
    return bytes_to_read;
}
```

Slika 7 surf_driver_read funkcija

3. Aplikacija

Na kraju treba napraviti aplikaciju koja će izvršavati komunikaciju sa drajverom i slati i dobijati vrednosti iz njega. U našem slučaju to je main.cpp iz specifikacije, kod je modifikovan tako da je iz njega izbačena createVector funkcija a zamenjena je sa **communicateWithDriver** koja omogućava komunikaciju.

Funkcija koristi sistemski poziv **open()** da otvori uređaj sa putanjom definisanom kao **DEVICE_PATH**. Drajver se otvara u režimu čitanja i pisanja.

Preko funkcije **memcpy()** se kopiraju vrednosti u bafer, što se radi i za osnovne parametre i za **pixels1D** i **_lookup2** nizove.

Podaci iz bafera se šalju kernel drajveru pomoću sistemskog poziva **write()**.

Proverava se da li je broj poslatih bajtova jednak ukupnoj veličini podataka. Ako broj ne odgovara, funkcija završava rad.

- Nakon slanja podataka, funkcija priprema bafer za čitanje podataka iz drajvera, gde će se čitati niz **index1D**.

- Sistemskim pozivom **read()** čitaju se podaci iz drajvera u bafer **readBuffer**. Ako dođe do greške, ispisuje se poruka i funkcija izlazi.

Ispisuju se svi pročitani podaci iz niza **index1D** kako bi se proverila tačnost podataka.

Na kraju, alocirani memorijski resursi se oslobađaju, a fajl deskriptor se zatvara pomoću **close()**.

```
void communicateWithDriver(double scale, double row, double col) {
    int fd = open(DEVICE_PATH, O_RDWR);
    if (fd < 0) {
        cerr << "Neuspešno otvaranje uređaja\n";
        return;
    }
    // Ukupna veličina bafera sada uključuje pixels1D i _lookup2
    size_t pixelsSize = _width * _height * sizeof(double);
    size_t lookup2Size = 40 * sizeof(double); // Veličina niza _lookup2
    size_t totalSize = sizeof(double) * 5 + pixelsSize + lookup2Size; // scale, row, col, SINE, COSE + pixels1D + _lookup2
    unsigned char* buffer = new unsigned char[totalSize];
    // Kopiraj podatke za scale, row, col, SINE, COSE
    memcpy(buffer, &scale, sizeof(double));
    memcpy(buffer + sizeof(double), &row, sizeof(double));
    memcpy(buffer + 2 * sizeof(double), &col, sizeof(double));
    memcpy(buffer + 3 * sizeof(double), &SINE, sizeof(double));
    memcpy(buffer + 4 * sizeof(double), &COSE, sizeof(double));
    // Kopiraj niz pixels1D u bafer
    memcpy(buffer + sizeof(double) * 5, pixels1D, pixelsSize);
    // Kopiraj niz _lookup2 u bafer
    memcpy(buffer + sizeof(double) * 5 + pixelsSize, _lookup2, lookup2Size);
    // Pošalji bafer kernelu
    ssize_t bytesSent = write(fd, buffer, totalSize);
    if (bytesSent < 0) {
        cerr << "Greška prilikom slanja bafera drajveru\n";
    }
    else if ((size_t)bytesSent != totalSize) {
        cerr << "Neispravan broj poslatih bajtova drajveru\n";
        delete[] buffer;
        close(fd);
        return;
    }
    // Sada ćemo čitati podatke iz drajvera (_index1D)
    int IndexSize = _IndexSize;
    int OriSize = _OriSize;
    // Pripremi bafer za čitanje podataka
    size_t totalReadSize = IndexSize * IndexSize * OriSize * sizeof(double);
    double* readBuffer = new double[IndexSize * IndexSize * OriSize];
    // Čitaj podatke iz drajvera
    ssize_t bytesRead = read(fd, readBuffer, totalReadSize);
    if (bytesRead < 0) {
        perror("Greška prilikom čitanja podataka iz drajvera");
        delete[] readBuffer;
        close(fd);
        return;
    }
    std::cout << "Pročitano " << bytesRead << " bajtova iz drajvera\n";
    // Obrada podataka (prikaz kao primer)
    for (size_t i = 0; i < bytesRead / sizeof(double); ++i) {
        std::cout << "readBuffer[" << i << "] = " << readBuffer[i] << std::endl;
    }
    // Oslobodi resurse
    delete[] readBuffer;
    close(fd);
}
```

Slika 8 communicateWithDriver funkcija

4.Dodatak

U kodu u komentarima su dodati mutexi za zaključavanje procesa, oni se koriste da se samo jedna funkcija izvršava u datom trenutku. Nakon završetka te funkcije mutex se oslobađa i prelazi se na sledeću.

Takođe dodate su static int __init surf_driver_init(void) i static void __exit surf_driver_exit(void) koje prave automatizaciju major broja preko kreiranje klase (class) i uređaja (device), i tako nije potrebno raditi ručno njegovo generisanje. Jedna funkcija je za inicijalizaciju a druga za brisanje, slične su kao init_module i cleanup_module.