İŞLETİM SİSTEMLERİ Dersi Dönem Projesi Raporu

MOTASEM YILDIZ

2221221380

Proje Tanıtımı:

ProCX, birden fazla programı aynı anda yönetmek, başlatmak ve sonlandırmak için geliştirilmiş bir sistemdir. Bu sistem, aynı anda birçok işlemi başlatmak, izlemek ve yönetmek için terminal tabanlı bir çözüm sunar. Aynı zamanda monitor thread'iyle işlemlerin durumunu gerçek zamanlı olarak takip eder ve message queue üzerinden bildirimler gönderir. ProCX kullanıcının, programları etkin bir şekilde izlemesini ve yönetmesini sağlar.

Proje Özellikleri:

1. Yeni Program Başlatma:
 - Kullanıcı, attached veya detached modda bir program başlatabilir.
 - Attached modda program, terminalle bağlantılı olarak çalışırken, detached modda program bağımsız olarak çalışır ve terminalden ayrılır.
 - Program başlatma işlemi, sistemdeki shared memory'ye kaydedilir.

2. Çalışan Programları Listeleme:
 - Sistem, şu anda çalışan tüm programları listeler.
 - Her bir programın PID'si, komutu, modu (attached veya detached), durumu (çalışıyor/sonlanmış) ve başlama zamanı görüntülenir.

3. Program Sonlandırma:
 - Kullanıcı, bir programı sonlandırmak için PID'yi girebilir.
 - SIGTERM sinyali gönderilerek işlem sonlandırılır ve durumu güncellenir.

Teknik Detaylar:

1. Shared Memory ve Semaphore Kullanımı:

Shared Memory (paylaşılan bellek), farklı işlemler arasında hızlı veri paylaşımı sağlayan önemli bir yapıdır. ProCX'te, tüm çalışan programların bilgileri (PID, komut, durum vb.) shared memory'de tutulur.

Semaphore, çoklu işlemlerin aynı anda paylaşılan kaynağa erişmesini kontrol etmek için kullanılır. Bu, race condition (yarış durumu) gibi problemleri engeller. ProCX'te her işlem semaphore kullanılarak kontrol edilir, böylece bir işlem paylaşılan belleğe eriştiğinde diğer işlemler bekler.

Kodda kullanılan semaphore yapısı:

```
// semafor lock 1 den 0
void lock_shared()
{
        if (g_sem == NULL) return;
        if (sem_wait(g_sem) == -1)
    {
        perror("sem_wait hatasi");
        exit(1);

    }
}

// semafor unlock 0 den 1
void unlock_shared()
{
        if (g_sem == NULL) return;
        if (sem_post(g_sem) == -1)
    {
        perror("sem_post hatasi");
        exit(1);

    }
}
```

2. Monitor Thread ve İşlem İzleme:

Monitor Thread, ProCX'teki merkezi bileşenlerden biridir. Bu thread, her iki saniyede bir çalışan programları kontrol eder ve durumu günceller. Ayrıca, detached modda çalışan programları sonlandırmaz, yalnızca attached modda olanları izler.

Her 2s'de bir kontrol yapılır, böylece işlemler hızlı bir şekilde izlenir.

```c
void *monitor_thread_func(void *arg)

{
        (void)arg;

        while (g_monitor_running)
    {

        sleep(2);

        // bu instancein child processleri için waitpid kullanacağız
        // başka instanceların processleri için kill(pid,0) ile kontrol
edeceğiz

        pid_t my_child_pids[50];
        int my_count = 0;

        lock_shared(); // kontrol etmeden önce semaforu kapatıyoruz bir
hata olmasın diye
        int pc = g_shared->process_count;

            for (int i = 0; i < pc && i < 50; i++){
                ProcessInfo *p = &g_shared->processes[i];
                // sadece aktif olan ve hala çalışan processleri kontrol
ediyoruz
            if (!p->is_active) continue;
                if (p->status != RUNNING) continue;

                if (p->owner_pid == g_self_pid)
            {
                // bu instancein child processleri
                my_child_pids[my_count++] = p->pid;

            }
            else{

                // Diğer instance'ların processleri için durum kontrolü
                if (kill(p->pid, 0) == -1 && errno == ESRCH)
                {
                    p->status = TERMINATED;
                    p->is_active = 0; // Temizle
                    printf("\n[MONITOR] Process %d sonlandı\n", p->pid);
                    printf("Seçiminiz: ");
                    fflush(stdout);
```

```
                    // diğer instaceleri bildir
                    send_ipc_message(CMD_STATUS_CHANGE, p->pid);

                }
            }
        }
            unlock_shared();

            // bu instance'in childlarini waitpid ile kontrol
        for (int k = 0; k < my_count; k++)
        {
            int status;
            pid_t ret = waitpid(my_child_pids[k], &status, WNOHANG);

            if (ret > 0){
                lock_shared();
                int pc2 = g_shared->process_count;
                for (int i = 0; i < pc2 && i < 50; i++){
                    ProcessInfo *p = &g_shared->processes[i];

                    if (p->is_active && p->pid == ret){
                        p->status = TERMINATED;
                        p->is_active = 0; // Temizle
                        break;
                    }

                }
                    unlock_shared();

                    printf("\n[MONITOR] Process %d sonlandı\n", ret);
                    printf("Seçiminiz: ");
                    fflush(stdout);

                    // diğer instancleri bildir
                send_ipc_message(CMD_STATUS_CHANGE, ret);

            }

        }

    }

        return NULL;
}
```

3. Message Queue ile Bildirimler:

Message Queue kullanılarak sistemdeki diğer instance'lara bildirimler gönderilir. Örneğin, bir işlem sonlandığında, bu durum IPC messages (işlem mesajları) aracılığıyla diğer sistemlere bildirilir.

```
// ipc message kuyruğa mesai gönderir
void send_ipc_message(IPCCommand cmd, pid_t target_pid)
{
    if (g_msgid == -1)
        return;

    Message msg;
    msg.msg_type = 1; // Tüm mesajlar için tip 1
    msg.command = cmd;
    msg.sender_pid = g_self_pid;
    msg.target_pid = target_pid;

    // msg_sz = sizeof(Message) - sizeof(long)
    if (msgsnd(g_msgid, &msg, sizeof(Message) - sizeof(long), IPC_NOWAIT) == -1)
    {
        // kuyruk doluysa geç
    }
}

// burda A process dan kuyruğa gönderdiğimiz mesaj B process kuyruktan alıp okuyacak
void *receiver_thread_func(void *arg)
{
```

Sonuçlar:

ProCX, çoklu işlem yönetimi ve izleme ihtiyaçlarını karşılamak için oldukça etkili bir çözümdür. Monitor thread ve message queue kullanarak, her bir işlem hızlı bir şekilde izlenebilir ve gerektiğinde sonlandırılabilir. Ayrıca, attached ve detached modlar arasında geçiş yapabilme, verimli bellek yönetimi ve hızlı bildirim sistemleri ile güçlü bir altyapıya sahiptir.