Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Λειτουργικά Συστήματα – Άσκηση 4

Των Προπτυχιακών Φοιτητών
Αρώνη Παναγιώτη Α.Μ. 03109083
Δανασή Παναγιώτη Α.Μ. 03109004
e-mail: el09004@mail.ntua.gr
panaro32@hotmail.com

1 Ασκήσεις:

1.1 Σχεδίαση χρονοδρομολογητή κυκλικής επαναφοράς στον χώρο χρήστη:

Ερωτήσεις:

1. Κατά την εγκατάσταση των ρουτινών εξυπηρέτησης των δύο αυτών σημάτων, έχουμε κάνει masked τα δύο αυτά σήματα. Με αυτόν τον τρόπο πετυχαίνουμε να μην γίνεται διακοπή από το δεύτερο σήμα όταν είμαστε στην ρουτίνα εξυπηρέτησης του ενός. Σε ένα πραγματικό χρονοδρομολογητή τα σήματα εκτελούνται σειριακά, κάτι παρόμοιο συμβαίνει και με τον δικό μας χρονοδρομολογητή καθώς αν είμαστε στην ρουτίνα εξυπηρέτησης του ενός σήματος, το άλλο γίνεται blocked και το σήμα αναμένει να επιστρέψουμε από την ρουτίνα οπότε και θα εκτελεστεί.

2. Το σήμα SIGCHLD που λαμβάνει κάθε φορά ο χρονοδρομολογητής μας περιμένουμε να αναφέρεται στην τρέχουσα διεργασία. Σε περίπτωση που για κάποιον αναπάντεχο εξωτερικό παράγοντα τερματιστεί κάποια από τις υπόλοιπες διεργασίες παιδιά, όταν έρθει ξανά η σειρά της, ο χρονοδρομολογητής μας θα δει ότι έχει πεθάνει, θα την θάψει και θα συνεχίσει κανονικά την δρομολόγηση με τις υπόλοιπες διεργασίες.

3. Ο χειρισμός δύο σημάτων χρειάζεται ώστε το ένα σήμα να επιτελεί την λειτουργία του timer που μετρά τα κβάντα χρόνου και το άλλο σήμα να κάνει το context switch σε περίπτωση αλλαγής κατάστασης μιας διεργασίας. Η χρονοδρομολόγηση θα μπορούσε να γίνει μόνο με την χρήση του σήματος SIGALRM όμως σε αυτήν την περίπτωση θα ήταν λιγότερο αποδοτικός μια και σε περίπτωση που μια διεργασία τερμάτιζε ή ανέστειλε την λειτουργία της δεν θα την άλλαζε αμέσως, αλλά θα έπρεπε να περιμένουμε να τελειώσει το κβάντο χρόνου για να αντικατασταθεί.

Περιγραφή υλοποίησης χρονοδρομολογητή:

Η υλοποίηση του χρονοδρομολογητή μας είναι βασισμένη στις συναρτήσεις χειρισμού δύο σημάτων, του σήματος SIGALRM και του σήματος SIGCHLD. Αρχικά δημιουργήσαμε μια συνδεδεμένη λίστα που περιέχει τις πληροφορίες για κάθε διεργασία που είναι σε κατάσταση waiting (στην προκειμένη υλοποίηση κρατάμε μόνο το PID της διεργασίας, μια και είναι αρκετό για τα ζητούμενα της άσκησης. Όμως η συνδεδεμένη λίστα μπορεί άνετα να γενικευτεί και να προστεθούν οποιεσδήποτε πληροφορίες μπορεί να θέλουμε για τις διεργασίες μας). Κάθε χρονική στιγμή τρέχει μόνο μία διεργασία. Αν αυτή η διεργασία πεθάνει ή μεταβεί σε κατάσταση waiting θα σταλεί αυτόματα ένα σήμα SIGCHLD στον χρονοδρομολογητή μας.
(Σε περίπτωση που μια διεργασία που είναι σε κατάσταση waiting τερματιστεί από κάποιον εξωτερικό παράγοντα, τότε με το που θα επιλεγεί από τον χρονοδρομολογητή μας, θα εκτελεστεί η συνάρτηση χειρισμού του SIGCHLD, οπότε η διεργασία θα θαφτεί κανονικά και θα συνεχιστεί η χρονοδρομολόγηση με τις υπόλοιπες).

Η συνάρτηση που χειρίζεται το σήμα αυτό (sigchld_handler) κάνει τα εξής:

- Αν το σήμα στάλθηκε επειδή πέθανε η διεργασία, απλά κάνει current την διεργασία που βρίσκεται στην κεφαλή της λίστας μας.
- Διαφορετικά αν απλά η διεργασία μπήκε σε κατάσταση waiting, τότε τοποθετεί την current διεργασία στο τέλος της λίστας μας και κάνει current την διεργασία που βρίσκεται στην κεφαλή της λίστας.

Έπειτα ορίζει να ξυπνήσει ο χρονοδρομολογητής μετά από «SCHED_TQ_SEC», θέτοντας το alarm και τέλος ενεργοποιεί την current διεργασία στέλνοντάς της SIGCONT.

Η συνάρτηση που χειρίζεται το σήμα SIGALRM (sigalrm_handler) απλά σταματάει την current διεργασία στέλνοντάς της σήμα SIGSTOP και αφήνει με αυτόν τον τρόπο όλη την δουλειά στην συνάρτηση sigchld_handler για να κάνει το context switch.

Η κεντρική συνάρτηση (main)

- Αρχικοποιεί την λίστα
- Κάνει fork όσα παιδιά είναι και οι διεργασίες που περάσαμε σαν όρισμα
- Αφού δημιουργήσει κάθε παιδί, αναστέλλει την λειτουργία του με SIGSTOP, και αφού σιγουρευτεί ότι έχει λάβει το σήμα και έχει σταματήσει (wait_for_ready_children) την προσθέτει στην λίστα με τις waiting διεργασίες.
- Εγκαθιστά τις συναρτήσεις χειρισμού των σημάτων
- Τέλος ζητάει να την ξυπνήσουν μετά από SCHED_TQ_SEC θέτοντας το alarm και ξεκινάει την πρώτη διεργασία.

```c
/*
 * Userspace Scheduler
 */

#include <errno.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>
#include <string.h>
#include <assert.h>

#include <sys/wait.h>
#include <sys/types.h>

#include "proc-common.h"
#include "request.h"

/* Compile-time parameters. */
#define SCHED_TQ_SEC 5                /* time quantum */
#define TASK_NAME_SZ 60               /* maximum size for a task's name */

/* Processes Library */
#include "queue.h"

queue q;
pid_t cur_pid;

/*
 * SIGALRM handler
 */
static void
sigalrm_handler(int signum)
{
    if (signum != SIGALRM) {
        fprintf(stderr, "Internal error: Called for signum %d, not SIGALRM\n",
            signum);
        exit(1);
    }

    printf("ALARM! %d seconds have passed.\n", SCHED_TQ_SEC);

    kill(cur_pid,SIGSTOP);
}

/*
 * SIGCHLD handler
 */
static void
sigchld_handler(int signum)
{
    int status;
    pid_t p;

    if (signum != SIGCHLD) {
        fprintf(stderr, "Internal error: Called for signum %d, not SIGCHLD\n",
            signum);
        exit(1);
    }
```

```
    p = waitpid(-1, &status, WUNTRACED | WNOHANG);
    if (p < 0) {
        perror("waitpid");
        exit(1);
    }

    if (p)
    {
        explain_wait_status(p, status);

        if (WIFEXITED(status) || WIFSIGNALED(status))
        {
            /* A child has died */
            if (queue_isempty(&q)) {
                fprintf(stderr, "Scheduler: No tasks. Exiting...\n");
                exit(1);
            }
            cur_pid = queue_remove(&q);
        }

        if (WIFSTOPPED(status))
        {
            /* A child has stopped due to SIGSTOP/SIGTSTP, etc */
            queue_add(&q,cur_pid);
            cur_pid = queue_remove(&q);
        }

        /* Setup the alarm again */
        if (alarm(SCHED_TQ_SEC) < 0) {
            perror("alarm");
            exit(1);
        }

        printf("Starting process with PID = %ld\n", (long)cur_pid);
        kill(cur_pid,SIGCONT);
    }
}

/* Install two signal handlers.
 * One for SIGCHLD, one for SIGALRM.
 * Make sure both signals are masked when one of them is running.
 */
static void
install_signal_handlers(void)
{
    sigset_t sigset;
    struct sigaction sa;

    sa.sa_handler = sigchld_handler;
    sa.sa_flags = SA_RESTART;
    sigemptyset(&sigset);
    sigaddset(&sigset, SIGCHLD);
    sigaddset(&sigset, SIGALRM);
    sa.sa_mask = sigset;
    if (sigaction(SIGCHLD, &sa, NULL) < 0) {
        perror("sigaction: sigchld");
        exit(1);
    }
```

```
    sa.sa_handler = sigalrm_handler;
    if (sigaction(SIGALRM, &sa, NULL) < 0) {
        perror("sigaction: sigalrm");
        exit(1);
    }

    /*
     * Ignore SIGPIPE, so that write()s to pipes
     * with no reader do not result in us being killed,
     * and write() returns EPIPE instead.
     */
    if (signal(SIGPIPE, SIG_IGN) < 0) {
        perror("signal: sigpipe");
        exit(1);
    }
}

int main(int argc, char *argv[])
{

    queue_init(&q);

    int nproc, i;
    pid_t p;

    /*
     * For each of argv[1] to argv[argc - 1],
     * create a new child process, add it to the process list.
     */
    for (i=1;i<argc;i++) {
        p = fork();
        if (p < 0) {
            /* fork failed */
            perror("fork");
            exit(1);
        }
        if (p == 0) {
            char executable[TASK_NAME_SZ];
            strcpy(executable,argv[i]);
            char *newargv[] = { executable, NULL, NULL, NULL };
            char *newenviron[] = { NULL };

            printf("I am PID = %ld\n", (long)getpid());
            printf("About to replace myself with the executable: %s...\n", executable);

            execve(executable, newargv, newenviron);

            /* execve() only returns on error */
            perror("execve");
            exit(1);
        }
        kill(p, SIGSTOP);
        wait_for_ready_children(1);

        // Add process to our struct
        queue_add(&q,p);
    }
```

```c
    nproc = argc-1; /* number of proccesses goes here */


    /* Install SIGALRM and SIGCHLD handlers. */
    install_signal_handlers();

    if (nproc == 0) {
        fprintf(stderr, "Scheduler: No tasks. Exiting...\n");
        exit(1);
    }

    queue_print(&q);

    /* Get first process */
    cur_pid = queue_remove(&q);

    /* Setup the alarm */
    if (alarm(SCHED_TQ_SEC) < 0) {
        perror("alarm");
        exit(1);
    }

    /* Start first process */
    kill(cur_pid,SIGCONT);

    /* loop forever  until we exit from inside a signal handler. */
    while (pause()) {
        /* Setup the alarm again */
        if (alarm(SCHED_TQ_SEC) < 0) {
            perror("alarm");
            exit(1);
        }
    }

    /* Unreachable */
    queue_free(&q);
    fprintf(stderr, "Internal error: Reached unreachable point\n");
    return 1;
}
```

```c
/*
*
*   queue.c
*
*/

#include <stdio.h>
#include <stdlib.h>

#include "queue.h"

void queue_init(queue *q)
{
    q->head=NULL;
    q->tail=NULL;
    q->size=0;
    return;
}

void queue_free(queue *q)
{
    while(q->head)
    {
        node *p=q->head;
        q->head=q->head->next;
        free(p);
    }
    return;
}

int queue_isempty(queue *q)
{
    return !(q->head);
}

void queue_add(queue *q,int x)
{
    node *p=(node*)malloc(sizeof(node));
    p->data=x;
    p->next=NULL;
    if(queue_isempty(q)) q->head=p;
    else q->tail->next=p;
    q->tail=p;
    q->size++;
    return;
}

int queue_remove(queue *q)
{
    if(queue_isempty(q))
    {
        fprintf(stderr,"Error: dequeue: Queue is empty\n");
        exit(1);
    }
    node *p=q->head;
    q->head=q->head->next;
    q->size--;
    if(queue_isempty(q)) q->tail=NULL;
    int x=p->data;
```

```
    free(p);
    return x;
}

int queue_delete(queue *q,int x)
{
    node *p=q->head;
    if(!p) return 0;
    if(p->data==x)
    {
        q->head=q->head->next;
        q->size--;
        if(queue_isempty(q)) q->tail=NULL;
        free(p);
        return 1;
    }
    while(p->next&&p->next->data!=x) p=p->next;
    if(p->next)
    {
        node *t=p->next;
        p->next=p->next->next;
        q->size--;
        if(q->tail==t) q->tail=p;
        free(t);
        return 1;
    }
    return 0;
}

void queue_print(queue *q)
{
    printf("QUEUE : [");
    node *p=q->head;
    while(p)
    {
        printf(" %d",p->data);
        p=p->next;
    }
    printf(" ]\n");
    return;
}
```