ΛΕΙΤΟΥΡΓΙΚΑ ΣΥΣΤΗΜΑΤΑ

Διαδιεργασιακή Επικοινωνία
Εργαστηριακές Ασκήσεις

Υλικό από:
Modern Operating Systems Laboratory Exercises, Shrivakan Mishra

Σύνθεση
Κ.Γ. Μαργαρίτης, Τμήμα Εφαρμοσμένης Πληροφορικής, Πανεπιστήμιο Μακεδονίας

1. Εκτελέστε τα προγράμματα με signals στις σελίδες 3 – 6. Τροποποιείστε τα προγράμματα:
1.1 Η γονική διεργασία μέσα στο while loop εκτελεί κάποια λειτουργία, πχ sleep(100) ή getchar().
Τι θα συμβεί όταν έρθει το σήμα τερματισμού της θυγατρικής διεργασίας;
1.2 Αλλαξτε τη σειρά των συναρτήσεων και τη θέση του for μέσα στο κώδικα της θυγατρικής διεργασίας. Αλλάζει κατι στην εκτέλεση; Κανετε την επικοινωνια με σήματα αμφιδρομη.

2. Υλοποιείστε τις συναρτήσεις που λείπουν από το πρόγραμμα της σελίδας 7.

3. Τα προγράμματα των σελίδων 8 – 12 λύνουν το πρόβλημα παραγωγού-καταναλωτή με διαφορετικές τεχνικές. Συγκρίνετε τα προγράμματα και τους μηχανισμούς IPC που χρησιμοποιούν. Προκαλέστε μεγάλες καθυστερήσεις σε παραγωγό ή καταναλωτή για να δείτε τι θα συμβεί. Τροποποιείστε τα προγράμματα ώστε η παραγωγή και η κατανάλωση να γίνεται σε ζεύγη δεδομένων (δηλ παραγωγή 2 δεδομένων, κατανάλωση 2 δεδομένων κοκ)

4. Συμπληρώστε το πρόγραμμα'πολλαπλών παραγωγών καταναλωτών στις σελίδες 13 – 15 με βάση τους μηχανισμούς IPC των σελίδων 8 – 12.

5.Τροποιείστε το πρόγραμμα στις σελίδες 16 - 17 όπως στα ερωτήματα 3 και 4.

6. Στις σελίδες 18 – 19 φαίνονται δύο εκδοχές δημιουργίας διοχέτευσης. Υλοποιείστε το πρώτο πρόγραμμα με τη λογική του δευτέρου και αντίστροφα.

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

/* When a SIGINT signal arrives, set this variable. */
int usr_interrupt = 0;

void synch_signal (int sig)
{
  usr_interrupt = 1;
}

/* The child process executes this function. */
void child_function (void)
{
  /* Perform initialization. */
  printf ("I'm here!!! My pid is %d.\n", (int) getpid ());

  /* Let parent know you're done. */
  kill (getppid (), SIGINT);

  /* Continue with execution. */
  printf ("Bye, now....\n");
  exit (0);
}
```

```
int main (void)
{
  pid_t child_id;

  /* Establish the signal handler. */
  signal (SIGINT, synch_signal);

  /* Create the child process. */
  child_id = fork();
  if (child_id == 0)
    child_function (); /* Does not return. */

  /* Busy wait for the child to send a signal. */
  while (!usr_interrupt)
    ;

  /* Now continue execution. */
  printf ("That's all, folks!\n");

  return 0;
}
```

```
/* sig_talk.c --- Example of how 2 processes can talk */
/* to each other using kill() and signal() */
/* We will fork() 2 process and let the parent send a few */
/* signals to it`s child  */

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

void sighup(); /* routines child will call upon sigtrap */
void sigint();
void sigquit();

main()
{ int pid;

  /* get child process */

   if ((pid = fork()) < 0) { /* unsuccesful fork */
        perror("fork");
        exit(1);
    }

   if (pid == 0)
     { /* child */
       signal(SIGHUP,sighup); /* set function calls */
       signal(SIGINT,sigint);
       signal(SIGQUIT, sigquit);
       for(;;); /* loop for ever */
     }
```

```
else /* parent */
     { /* pid hold id of child */
       printf("\nPARENT: sending SIGHUP\n\n");
       kill(pid,SIGHUP);
       sleep(3); /* pause for 3 secs */
       printf("\nPARENT: sending SIGINT\n\n");
       kill(pid,SIGINT);
       sleep(3); /* pause for 3 secs */
       printf("\nPARENT: sending SIGQUIT\n\n");
       kill(pid,SIGQUIT);
       sleep(3);
     }
}


void sighup()

{  signal(SIGHUP,sighup); /* reset signal */
   printf("CHILD: I have received a SIGHUP\n");
}

void sigint()

{  signal(SIGINT,sigint); /* reset signal */
   printf("CHILD: I have received a SIGINT\n");
}

void sigquit()

{ printf("My DADDY has Killed me!!!\n");
  exit(0);
}
```

```
#define N 100                                   /* number of slots in the buffer */
typedef int semaphore;                          /* semaphores are a special kind of int */
semaphore mutex = 1;                            /* controls access to critical region */
semaphore empty = N;                            /* counts empty buffer slots */
semaphore full = 0;                             /* counts full buffer slots */

void producer(void)
{
     int item;

     while (TRUE) {                             /* TRUE is the constant 1 */
          item = produce_item( );               /* generate something to put in buffer */
          down(&empty);                         /* decrement empty count */
          down(&mutex);                         /* enter critical region */
          insert_item(item);                    /* put new item in buffer */
          up(&mutex);                           /* leave critical region */
          up(&full);                            /* increment count of full slots */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* infinite loop */
          down(&full);                          /* decrement full count */
          down(&mutex);                         /* enter critical region */
          item = remove_item( );                /* take item from buffer */
          up(&mutex);                           /* leave critical region */
          up(&empty);                           /* increment count of empty slots */
          consume_item(item);                   /* do something with the item */
     }
}
```

```
#include <pthread.h>
#include <stdio.h>
#include <semaphore.h>
#define BUFF_SIZE 4
#define SHARED 1

char buffer[BUFF_SIZE];
int nextIn = 0;
int nextOut = 0;

sem_t empty_slots;
sem_t full_slots;

void Put(char item) {

        sem_wait(&empty_slots);
        buffer[nextIn] = item;
        nextIn = (nextIn + 1) % BUFF_SIZE;
        printf("Producing %c ...\n", item);
        sem_post(&full_slots);

}

void * Producer() {

        int i;
        for(i = 0; i < 10; i++)
                Put((char)('A'+ i % 26));

}
```

```
void Get() {

        int item;
        sem_wait(&full_slots);
        item = buffer[nextOut];
        nextOut = (nextOut + 1) % BUFF_SIZE;
        printf("Consuming %c ...\n", item);
        sem_post(&empty_slots);

}

void * Consumer() {

        int i;
        for(i = 0; i < 10; i++)
                Get();

}


main() {

        pthread_t tid1, tid2;

        sem_init(&empty_slots, SHARED, 4);
        sem_init(&full_slots, SHARED, 0);
        pthread_create(&tid1, NULL, Producer, NULL);
        pthread_create(&tid2, NULL, Consumer, NULL);
        pthread_join(tid1, NULL);
        pthread_join(tid2, NULL);

}
```

```
/* Producer/consumer program illustrating conditional variables */
#include<pthread.h>
#include <stdio.h>

#define BUF_SIZE 3                                              /* Size of shared buffer */

int buffer[BUF_SIZE];                                           /* shared buffer */
int add=0;                                                      /* place to add next element */
int rem=0;                                                      /* place to remove next element */
int num=0;                                                      /* number elements in buffer */
pthread_mutex_t m=PTHREAD_MUTEX_INITIALIZER     ;       /* mutex lock for buffer */
pthread_cond_t c_cons=PTHREAD_COND_INITIALIZER; /* consumer waits on this cond var */
pthread_cond_t c_prod=PTHREAD_COND_INITIALIZER; /* producer waits on this cond var */

void *producer(void *param);
void *consumer(void *param);

main (int argc, char *argv[])
{
        pthread_t tid1, tid2;           /* thread identifiers */
        int i;

        /* create the threads; may be any number, in general */
        if (pthread_create(&tid1,NULL,producer,NULL) != 0) {
                fprintf (stderr, "Unable to create producer thread\n");
                exit (1);
        }
        if (pthread_create(&tid2,NULL,consumer,NULL) != 0) {
                fprintf (stderr, "Unable to create consumer thread\n");
                exit (1);
        }
        /* wait for created thread to exit */
        pthread_join(tid1,NULL);
        pthread_join(tid2,NULL);
        printf ("Parent quiting\n");
}
```

```
/* Produce value(s) */
void *producer(void *param)
{
        int i;
        for (i=1; i<=20; i++) {                                 /* Insert into buffer */
                pthread_mutex_lock (&m);
                if (num > BUF_SIZE) exit(1);                    /* overflow */
                while (num == BUF_SIZE)                         /* block if buffer is full */
                        pthread_cond_wait (&c_prod, &m);        /* if executing here, buffer not full so add element */
                buffer[add] = i;
                add = (add+1) % BUF_SIZE;
                num++;
                pthread_mutex_unlock (&m);
                pthread_cond_signal (&c_cons);
                printf ("producer: inserted %d\n", i);  fflush (stdout);
        }
        printf ("producer quiting\n");  fflush (stdout);
}

/* Consume value(s); Note the consumer never terminates */
void *consumer(void *param)
{
        int i;
        while (1) {
                pthread_mutex_lock (&m);
                if (num < 0) exit(1);                           /* underflow */
                while (num == 0)                                /* block if buffer empty */
                        pthread_cond_wait (&c_cons, &m);        /* if executing here, buffer not empty so remove element */
                i = buffer[rem];
                rem = (rem+1) % BUF_SIZE;
                num--;
                pthread_mutex_unlock (&m);
                pthread_cond_signal (&c_prod);
                printf ("Consume value %d\n", i);  fflush(stdout);
        }
}
```

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

pthread_mutex_t mtx;
pthread_cond_t cond;

int how_many = 10;
int pool = 0;

void * producer(void * ptr)
{
    while (how_many > 0)
    {
        pthread_mutex_lock(&mtx);
        printf("producer: %d\n", how_many);
        pool = how_many;
        how_many--;
        pthread_mutex_unlock(&mtx);
        pthread_cond_signal(&cond);
    }
    pthread_exit(0);
}
```

```
void * consumer(void * ptr)
{
    while (how_many > 0)
    {
        pthread_mutex_lock(&mtx);
        pthread_cond_wait(&cond, &mtx);
        printf("consumer: %d\n", pool);
        pool = 0;
        pthread_mutex_unlock(&mtx);
    }
    pthread_exit(0);
}

int main(int argc, char ** argv)
{
    pthread_t prod, cons;
    pthread_mutex_init(&mtx, 0);
    pthread_cond_init(&cond, 0);
    pthread_create(&cons, 0, consumer, 0);
    pthread_create(&prod, 0, producer, 0);
    pthread_join(prod, 0);
    pthread_join(cons, 0);
    pthread_cond_destroy(&cond);
    pthread_mutex_destroy(&mtx);
    return 0;
}
```

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

#define BUFF_SIZE   5       /* total number of slots */
#define NP          3       /* total number of producers */
#define NC          3       /* total number of consumers */
#define NITERS      4       /* number of items produced/consumed */

typedef struct {
    int buf[BUFF_SIZE];   /* shared var */
    int in;               /* buf[in%BUFF_SIZE] is the first empty slot */
    int out;              /* buf[out%BUFF_SIZE] is the first full slot */
    sem_t full;           /* keep track of the number of full spots */
    sem_t empty;          /* keep track of the number of empty spots */
    sem_t mutex;          /* enforce mutual exclusion to shared data */
} sbuf_t;

sbuf_t shared;
```

```
void *Producer(void *arg)
{
    int i, item, index;

    index = (int)arg;

    for (i=0; i < NITERS; i++) {

        /* Produce item */
        item = i;
        printf("[P%d] Producing %d ...\n", index, item); fflush(stdout);

        /* Write item to buf */

        /* If there are no empty slots, wait */
        sem_wait(&shared.empty);
        /* If another thread uses the buffer, wait */
        sem_wait(&shared.mutex);
        shared.buf[shared.in] = item;
        shared.in = (shared.in+1)%BUFF_SIZE;
        /* Release the buffer */
        sem_post(&shared.mutex);
        /* Increment the number of full slots */
        sem_post(&shared.full);

        /* Interleave  producer and consumer execution */
        if (i % 2 == 1) sleep(1);
    }
    return NULL;
}
```

```
void *Consumer(void *arg)
{
    /* Fill in the code here */
}


int main()
{
    pthread_t idP, idC;
    int index;

    sem_init(&shared.full, 0, 0);
    sem_init(&shared.empty, 0, BUFF_SIZE);

    /* Insert code here to initialize mutex*/

    for (index = 0; index < NP; index++)
    {
       /* Create a new producer */
       pthread_create(&idP, NULL, Producer, (void*)index);
    }

    /* Insert code here to create NC consumers */

    pthread_exit(NULL);
}
```

```
#include <pthread.h>
#include <mqueue.h>

int main(void)
{
    mqd_t messageQueueDescr;

    /* Create the message queue for sending information between tasks. */
    messageQueueDescr = mq_open(messageQueuePath,
                                          (O_CREAT | O_EXCL | O_RDWR));

    /* Create the producer task using the default task attributes. Do not
     * pass in any parameters to the task. */
    pthread_create(&producerTaskObj, NULL, (void *)producerTask, NULL);

    /* Create the consumer task using the default task attributes. Do not
     * pass in any parameters to the task. */
    pthread_create(&consumerTaskObj, NULL, (void *)consumerTask, NULL);

    /* Allow the tasks to run. */
    pthread_join(producerTaskObj, NULL);
    pthread_join(consumerTaskObj, NULL);

    return 0;
}
```

```
void producerTask(void *param)
{
    mqd_t messageQueueDescr;
    msgbuf_t msg;
    messageQueueDescr = mq_open(messageQueuePath, O_WRONLY);
    while (1)
    {
            /* Create message msg */
            /* Send message */
            mq_send(messageQueueDescr, &msg.buf, sizeof(msgbuf_t), 0);
        }
    }
}
void consumerTask(void *param)
{
    mqd_t messageQueueDescr;
    msgbuf_t rcvMsg;
    messageQueueDescr = mq_open(messageQueuePath, O_RDONLY);
    while (1)
    {
        /* Wait for a new message. */
        mq_receive(messageQueueDescr, &rcvMsg.buf, sizeof(msgbuf_t),
                                                              NULL);
        /* Cosnume message msg */
    }
}
```

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

void write_data (FILE * stream)
{
  int i;
  for (i = 0; i < 100; i++)
    fprintf (stream, "%d\n", i);
  if (ferror (stream))
  {
    fprintf (stderr, "Output to stream failed.\n");
    exit (EXIT_FAILURE);
  }
}

int main (void)
{
  FILE *output;

  output = popen ("more", "w");
  if (!output)
  {
    fprintf (stderr,"incorrect parameters or too many files.\n");
    return EXIT_FAILURE;
  }
  write_data (output);
  if (pclose (output) != 0)
  {
    fprintf (stderr,"Could not run more or other error.\n");
  }
  return EXIT_SUCCESS;
}
```

```
#include <stdio.h>
#include <string.h>
#include <sys/wait.h>
int main(int argc, char *argv[])
{ int p[2];
  int i, pid, status;
  char buffer[20];
    pipe(p);                          /* setting up the pipe   */
    if ((pid = fork()) == 0)
    /* in child */
    {  close(p[1]);                   /* child closes p[1]     */
       while ((i = read(p[0], buffer, 8)) != 0)
       { buffer[i] = '\0';            /* string terminator     */
         printf("%d chars :%s: received by child\n", i, buffer);
       }
       close(p[0]);
       exit(0);                     /* child terminates      */
    }
    /* in parent */
    close(p[0]);
    /* parent writes p[1]    */
    write(p[1], "Hello there, from me?", sizeof("Hello there, from me?"));
    write(p[1], "How are you? - Ercal", sizeof("How are you? - Ercal"));
    close(p[1]);                      /* finished writing p[1] */
    while (wait(&status)!=pid);       /* waiting for pid       */
    if (status == 0)  printf("child finished\n");
    else printf("child failed\n");
    return(0);
}
```