Case study

Read in and write to

1. Problem description:
   A file(counter.txt) contains an integer number.
   A process can read the number and increase it and then write the result back tofile.
   How about the multi-processes read and write parallel?
   How to implement multi-threads reading and writing by using binary semaphore or mutexes?

2. A process read from and write back to a file

```
#include <sys/types.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <stdlib.h>
```

```
#include <unistd.h>
#define COUNTER_FILE "counter.txt"
#define SLEEP_TIME 2
int read_counter(char filename[]) {
  FILE *fp;
  int counter;
  if ((fp=fopen(filename,"r"))==NULL){
    perror(filename);
    exit(1);
  }
  if(fscanf(fp,"%d",&counter)!=1) {
    printf("integer not found.\n");
    exit(1);
  }
  if(fclose(fp)!=0) {
    perror(filename);
    exit(1);
  }
  return counter;
}
int write_counter(char filename[],int counter)
{
  FILE *fp;
```

```
  if ((fp=fopen(filename,"w"))==NULL){
    perror(filename);
    exit(1);
  }
  if(fprintf(fp,"%d\n",counter)<0) {
    printf("integer not written.\n");
    exit(1);
  }
  if(fclose(fp)!=0) {
    perror(filename);
    exit(1);
  }
  return 0;
}
int main(int argc, char *argv[])
{
  int counter;
  int loop_nr, i;
  if (argc==1 ||
      sscanf(argv[1],"%d",&loop_nr)!=1) {
    printf("usage: %s nr\n\t- where nr is
                    an integer\n",argv[0]);
    exit(1);
```

```
  }
  printf("Looping %d times\n", loop_nr);
  for (i=0; i<loop_nr; i++) {
    counter=read_counter(COUNTER_FILE);
    printf("read counter: %d\n", counter);
    sleep(SLEEP_TIME);
    counter++;
    write_counter(COUNTER_FILE,counter);
    printf("write counter: %d\n", counter);
    sleep(SLEEP_TIME);
  }
}
```

3. Multi-processes read and write parallel

```
int main(int argc, char *argv[])
{
  int counter;
  int loop_nr, i;
  if (argc==1 ||
      sscanf(argv[1],"%d",&loop_nr)!=1){
    printf("usage: %s nr\n\t- where nr
```

```
                is an integer\n",argv[0]);
    exit(1);
  }
  for (i=0; i<loop_nr; i++) {
    pid_t pid;
    pid = fork();
    if (pid <0){
      printf("unable to fork.\n");
    }else{
      if (pid > 0){
counter=read_counter(COUNTER_FILE);
printf("parent read counter: %d\n", counter);
sleep(SLEEP_TIME);
counter++;

write_counter(COUNTER_FILE,counter);
printf("parent write counter: %d\n", counter);
sleep(3);
      }else {
counter=read_counter(COUNTER_FILE);
printf("child read counter: %d\n", counter);
sleep(SLEEP_TIME);
counter++;
```

```
write_counter(COUNTER_FILE,counter);
printf("child write counter: %d\n", counter);
sleep(SLEEP_TIME);
      }
    }
  }
}
```

4. Multi-threads read and write with binary semaphores

```
sem_t bin_sem;
int counter;
void *thread_function(void *arg)
{
  sem_wait(&bin_sem);
  while(1){
    counter=read_counter(COUNTER_FILE);
    printf("read counter: %d\n", counter);
    counter++;
    write_counter(COUNTER_FILE,counter);
```

```
    printf("write counter: %d\n", counter);
    sem_wait(&bin_sem);
  }
  pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
 int res;
 int loop_nr, i;
 pthread_t thread;
 if (argc==1 ||
  sscanf(argv[1],"%d",&loop_nr)!=1){
    printf("usage: %s nr\n\t- where nr
              is an integer\n",argv[0]);
    exit(EXIT_FAILURE);
 }
 res = sem_init(&bin_sem, 0, 0);
 if(res != 0){
   printf("Failed to initialize semaphore\n");
   exit(EXIT_FAILURE);
 }
 res=pthread_create(&thread, NULL, thread_func
```

```
 if(res!=0){
   perror("Thread creation failure\n");
   exit(EXIT_FAILURE);
 }
 printf("Looping %d times\n", loop_nr);
 for(i=0; i<loop_nr; i++){
  sleep(1);
  sem_post(&bin_sem);
 }
 pthread_detach(thread);
 sem_destroy(&bin_sem);
 exit(EXIT_SUCCESS);
}
```

Dinning Philosophers

1. Problem description:
   N philosophers sitting in a round table to eat spaghetti.
   Each Philosopher either eating or thinking.
   Each Philosopher share the left and right fork with the neighbour.
   Assume that need two forks at same time to eat spaghetti.

2. Create an array of threads as philosophers

```
#include <pthread.h>
#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>
#include <unistd.h>

#define NMAX 50
```

```
pthread_t threads[NMAX];

typedef struct{
        int number;
        }phil_data;

phil_data phil_data_tab[NMAX];

void *phil(void *arg)
{
  int my_number;
  my_number= ((phil_data*)arg)-> number;
  printf("Thread %d started\n", my_number);
}
int main(int *agrc, char *argv[])
{
  int i, n;
  n= atoi(argv[1]);

  for(i= 0; i<n; i++){
     phil_data_tab[i].number = i;
     pthread_create(&(threads[i]), 0, phil,
```

```
                                &(phil_data_tab[i]));
    }
    pthread_detach(threads[i]);
    pthread_exit(0);
  }
```

3. use less data structure to create an array of threads

```
#include <pthread.h>
#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>
#include <unistd.h>

#define NMAX 50

pthread_t threads[NMAX];

void *new_thread(void *arg)
{
  int my_number;
```

```
  my_number = *(int*)arg;
  printf("Thread %d started\n", my_number);
}
int main(int argc, char *argv[])
{
  int i, n;
  if(argc < 2)
    printf("Incorrect number of argument");
  n = atoi(argv[1]);

  for(i=0; i<n; i++){
    pthread_create(&threads[i], NULL,
                           new_thread, &i);
  }
  pthread_detach(threads[i]);
  pthread_exit(0);
}
```

4. How to grab the forks

```
#define NMAX 50
```

```
/* declare an array of forks, returns 1 if
                forks available, 0 if not */
static int forks[NMAX];
pthread_mutex_t mutex;

/*this function grab a single fork*/
int grab(int fork_number)
{
  pthread_mutex_lock(&mutex);
/* wait for fork to become available*/
  while(forks[fork_number]!=1){
   printf("philosopher %d fail to grab fork
                %d.\n",my_number,fork_number);
   pthread_mutex_unlock(&mutex);
   usleep(THINK_TIME);
   pthread_mutex_lock(&mutex);
  }
/* grab fork, set it as unavailable*/
  forks[fork_number]= 0;
  pthread_mutex_unlock(&mutex);
}

/*this function release a single fork*/
```

```
int release(int fork_number)
{/* release the fork, set it available*/
  forks[fork_number]= 1;
}
```

5. Define philosopher routine

```
#define EATING_TIME 3
#define THINKING_TIME 5
sem_t bin_sem[NMAX];
void *phil(void *arg)
{
 int my_number;
 int fork_number;
 my_number = *(int*)arg;

/*grab both forks at the same time*/
 sem_wait(&bin_sem[my_number]);
 while(grab(my_number)){
  grab((my_number+1)%NMAX);
 }
 printf("Philosopher %d starts to eat.\n",
```

```
                        my_number);
 sleep(EAT_TIME);
 release(my_number);
 release((my_number+1)%NMAX);
 printf("Philosopher %d starts to think.\n",
                         my_number);
 sleep(THINK_TIME);
}
```

6. Implement the eating schedule

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <sys/types.h>
int main(int *agrc, char *argv[])
{
 int res;
 pthread_t threads[NMAX];
 int i, n;
```

```
 n= atoi(argv[1]);
for (i=0; i<n; i++){
 forks[i]=1; /* initialize forks*/
 res= sem_init(&bin_sem[i], 0, 0);
 if (res!=0){
  perror("semaphore failed to initialized.\n")
  exit(EXIT_FAILURE);
 }
 res = pthread_create(&threads[i], 0,
                                  phil, &i);
 if (res!=0){
  perror("thread creation failed.\n");
  exit(EXIT_FAILURE);
 }
 sem_post(&bin_sem[i]);
}
pthread_detach(threads[i]);
sem_destroy(&bin_sem[i]);
exit(EXIT_SUCCESS);
}
```

THE END