CSC552 – Advanced UNIX Programming

Classic IPC Problems

Dr. L. Frye
Kutztown University

Dining Philosophers

Image from Wikipedia

Key Question

- Can you write a program for each philosopher that does what it is supposed to do and never gets stuck?

Obvious Solution

```
#define  N  5              /* number of philosophers */

void philosophers(int i)     /* i: philosopher #, from 0 to 4 */
{
    while (TRUE)  {
        think ();                    /* philosopher is thinking */
        take_fork(i);                /* take left fork */
        take_fork((i+1) % N);        /* take right fork */
        eat();                       /* eat spaghetti */
        put_fork(i);                 /* put left fork back on the table */
        put_fork((i+1) % N);         /* put right fork back on table */
    }
}   /* end philosophers */
```

Questions on Solution

- Do you see any problems with this solution?
  - How could this be prevented?
  - Do you see any problems with this new solution?
    - Any solutions to prevent this?
  - Another solution – binary semaphores?
    - What is the problem with this solution?

Good Solution

- classics/diningPhil.c

- If time – implement as class
- If no time – look at code in example (diningPhil.c)

Readers and Writers Problem

- Models DB access
- Airline reservation system

- Two strategies
  - Strong reader synchronization
  - Strong writer synchronization

One Solution

```
typedef  int  semaphore;
semaphore mutex = 1;   /* controls access to 'rc' */
semaphore db = 1;      /* controls access to DB */
int  rc = 0;           /* # of processes reading or
  wanting to */
```

```
void reader()   {
    while (TRUE)  {
        down(&mutex);      /* exclusive access rc */
        rc = rc + 1;      /* one more reader */
         /* if first reader, get access to DB */
        if (rc == 1) down(&db);
        up(&mutex);       /* release excl access rc */
        read_data_base();
        down(&mutex);      /* get excl access to rc */
        rc = rc – 1;        /* one less reader */
         /* if last reader, release access to DB */
        if (rc == 0) up(&db);
        up(&mutex);          /* release excl access */
        use_data_read();
    }
} /* end reader */
```

```
void writer()  {
    while (TRUE) {
        think_up_data();  /* noncritical section */
        down(&db);      /* get exclusive access */
        write_data_base();
        up(&db);          /* release excl access */
    }
}   /* end writer */
```

Questions on Solution

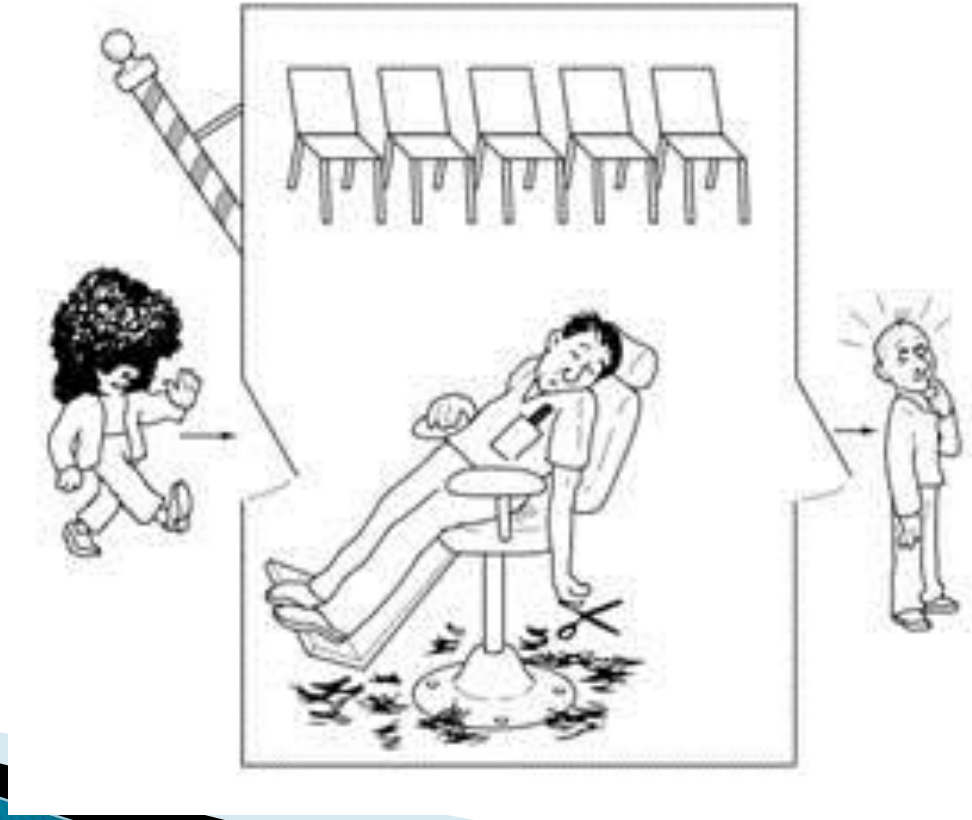
- Why is it necessary to have exclusive access to rc?
  - Subtle decision in this solution.
- What would happen when a writer arrives?
  - Any thoughts on how to prevent this situation?

Readers and Writers with Threads

- Read-write locks
  - Type: pthread_rwlock_t
  - pthread_rwlock_init()
  - pthread_rwlock_destroy()
  - Acquiring locks
    - pthread_rwlock_rdlock()
    - pthread_rwlock_tryrdlock()
    - pthread_rwlock_wrlock()
    - pthread_rwlock_trywrlock()
  - pthread_rwlock_unlock()

Sleeping Barber

- One barber
- One barber chair
- *n* chairs for waiting customers



Problem

- Program the barber and the customers without getting into any race conditions.

- Design solution BEFORE look at code!

One Solution

```
#define  CHAIRS  5        /* # chairs waiting customers */

typedef  int  semaphore;
semaphore customers = 0;   /* # customers waiting */
semaphore barbers = 0;     /* # barbers waiting */
semaphore mutex = 1;       /* for mutual exclusion */
int waiting = 0;           /* customers are waiting */
```

```
void barber()
{
   while (TRUE) {
        down(customers);  /* go to sleep if # custs = 0 */
        down(mutex);       /* acquire access to waiting */
        waiting = waiting - 1; /* decrement count of \
                                   waiting customers */
        up(barbers);      /* one barber ready to cut hair */
        up(mutex);            /* release waiting */
        cut_hair();           /* noncritical section */
   }
}   /* end barber */
```

```
void customer()
{
   down(mutex);              /* enter critical section */
   if (waiting < CHAIRS) {   /* if no free chairs, leave */
       waiting = waiting + 1;    /* increment count of
                                    waiting customers */
       up(customers);  /* wake up barber if necessary */
       up(mutex);            /* release access to waiting */
       down(barbers);        /* go to sleep if
                                    # free barbers=0 */
       get_haircut();
   }
   else {
       up(mutex);            /* shop is full, don't wait */
   }
}   /* end customer */
```