# Classical Synchronization Problems



## Classical mutual exclusion problems

- Bounded(N places)-Buffer
- Readers and Writers
- Dining-Philosophers
- Sleeping Barber
- Dining-Philosophers Problem Solution using monitors



## Implementation of Producer-consumer Shared Bounded-Buffer Problem Using Semaphore

- *Each of N* buffer places can hold one data item
- Implementation:
  - Use binary semaphore mutex to establish mutual exclusion on buffer update, initialized to 1
  - Use a multi-value semaphore full to implement item consumption, initialized to 0
  - Use a multi-value semaphore empty to implement item production, initialized N.



## Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
do {
     //   produce an item
   wait (empty); // queued if 0
   wait (mutex);
      //  add the item to the  buffer
    signal (mutex);
    signal (full); //allow consumer to consume if any
} while (true);
```



## Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
do {
   wait (full); //queue if 0
   wait (mutex);
      //  remove an item from  buffer
    signal (mutex);
    signal (empty); //allow producer to produce, if any
       //  consume the removed item
} while (true);
```



## Implement Readers-Writers Problem using Semaphore

- A data set is shared among a number of concurrent reader and writer processes
  - Readers – only read the data set; they do not perform any updates
  - Writers – write the data item to be read by the readers
- Design algorithm:
  - multiple readers can read an item, if exist, concurrently with no protection
  - Writer(s) can only write data item in mutual exclusion
  - A writer and a reader can write and read in mutual exclusion
- Modeling Shared Data
  - Data set: item
  - Semaphore mutex initialized to 1
  - Semaphore wrt initialized to 1
  - Integer readcount is readers shared memory, initialized to 0: it counts number of readers in the process of reading.

## Readers-Writers Problem (Cont.)

- writer process: should write only if there is no active reader

```
do {
    wait (wrt) ; // no limit on number of items

      //   writing item is performed

    signal (wrt) ;
} while (true)
```



## Readers-Writers Problem (Cont.)

- The structure of a reader process

```
do {
    wait (mutex) ;
    readcount ++ ;
    if (readercount == 1)  wait (wrt) ;
    signal (mutex)

        // reading item is performed

    wait (mutex) ;
    readcount  - - ;
    if redacount   == 0)  signal (wrt) ;
    signal (mutex) ;
} while (true)
```



## Dining-Philosophers Problem: 5 philosopher dine and think


5 Chinese philosophers dine and think randomly.

- Modeling: functions: think(), eat(), take_fork(), put_fork()
  - Share Data set:
    - Bowl of rice
    - 5 chopsticks: Semaphore fork [5], initialized to 1



## Dining Philosophers: First Try

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

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

Is this solution  correct? No control over the state of the forks!



## Dining Philosophers: Correct Try: Control over the state of the foks

```
#define N           5                 /* number of philosophers */
#define LEFT        (i+N−1)%N         /* number of i's left neighbor */
#define RIGHT       (i+1)%N           /* number of i's right neighbor */
#define THINKING    0                 /* philosopher is thinking */
#define HUNGRY      1                 /* philosopher is trying to get forks */
#define EATING      2                 /* philosopher is eating */
typedef int semaphore;                /* semaphores are a special kind of int */
int state[N];                         /* array to keep track of everyone's state */
semaphore mutex = 1;                  /* mutual exclusion for critical regions */
semaphore s[N];                       /* one semaphore per philosopher */

void philosopher(int i)               /* i: philosopher number, from 0 to N−1 */
{
     while (TRUE) {                   /* repeat forever */
          think( );                   /* philosopher is thinking */
          take_forks(i);              /* acquire two forks or block */
          eat( );                     /* yum-yum, spaghetti */
          put_forks(i);               /* put both forks back on table */
     }
}
```

Solution to dining philosophers problem (part 1)

Note that mutex controls all CSs; S[i] are initially set to 0



## Dining Philosophers

```
void take_forks(int i)                /* i: philosopher number, from 0 to N−1 */
{
     down(&mutex);                    /* enter critical region */
     state[i] = HUNGRY;               /* record fact that philosopher i is hungry */
     test(i);                         /* try to acquire 2 forks */
     up(&mutex);                      /* exit critical region */
     down(&s[i]);                     /* block if forks were not acquired */
}

void put_forks(i)                     /* i: philosopher number, from 0 to N−1 */
{
     down(&mutex);                    /* enter critical region */
     state[i] = THINKING;             /* philosopher has finished eating */
     test(LEFT);                      /* see if left neighbor can now eat */
     test(RIGHT);                     /* see if right neighbor can now eat */
     up(&mutex);                      /* exit critical region */
}

void test(i)                          /* i: philosopher number, from 0 to N−1 */
{
     if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
          state[i] = EATING;
          up(&s[i]);
     }
}
```

Solution to dining philosophers problem (part 2)

## Sleeping Barber



## The Sleeping Barber Problem (2)

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

typedef int semaphore;              /* use your imagination */

semaphore customers = 0;            /* # of customers waiting for service */
semaphore barbers = 0;              /* # of barbers waiting for customers */
semaphore mutex = 1;                /* for mutual exclusion */
int waiting = 0;                    /* customers are waiting (not being cut) */

void barber(void)
{
    while (TRUE) {
        down(&customers);           /* go to sleep if # of customers is 0 */
        down(&mutex);               /* acquire access to 'waiting' */
        waiting = waiting – 1;      /* decrement count of waiting customers */
        up(&barbers);               /* one barber is now ready to cut hair */
        up(&mutex);                 /* release 'waiting' */
        cut_hair( );                /* cut hair (outside critical region) */
    }
}


void customer(void)
{
    down(&mutex);                   /* enter critical region */
    if (waiting < CHAIRS) {         /* if there are 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 # of free barbers is 0 */
        get_haircut( );             /* be seated and be serviced */
    } else {
        up(&mutex);                 /* shop is full; do not wait */
    }
}
```

waiting is shared data item.



## Problems with Semaphores

- Correct use of semaphore operations is not easy.
- Omitting of wait (mutex) or signal (mutex) or both is cause of incorrect solutions.



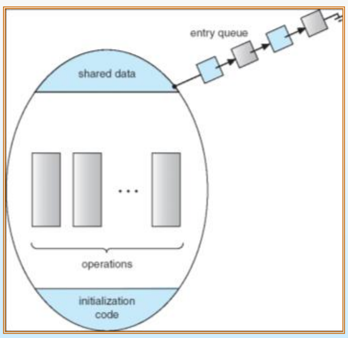
## Monitors: A higher level synchronization construct

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Only one process may be active within the monitor at a time

```
monitor monitor-name
{
  // shared variable declarations
  procedure P1 (…) { …. }
        …
  procedure Pn (…) {……}
   Initialization code ( ….) { … }
        …
  }
}
```



## Schematic view of a Monitor

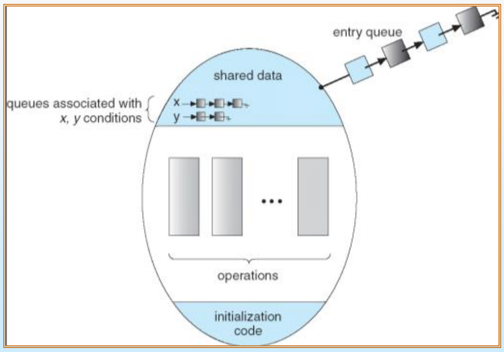





## Condition Variables and dining philosophers

- condition x, y;
- Two operations on a condition variable:
  - x.wait () – a process that invokes the operation is suspended.
  - x.signal () – resumes one of processes (if any) that invoked x.wait ()

## Monitor with Condition Variables






## Solution to Dining Philosophers: Monitor Solution

```
monitor DiningPhilosopher
  {
    enum { THINKING; HUNGRY, EATING) state [5] ;
    condition self [5];
    void pickup (int i) {
         state[i] = HUNGRY;
         test(i);
         if (state[i] != EATING) self [i].wait;
    }
     void putdown (int i) {
         state[i] = THINKING;
              // test left and right neighbors
          test((i + 4) % 5);
          test((i + 1) % 5);
     }
```



## Monitor Solution to Dining Philosophers (cont)

```
void test (int i) {
      if ( (state[(i + 4) % 5] != EATING) &&
      (state[i] == HUNGRY) &&
      (state[(i + 1) % 5] != EATING) ) {
          state[i] = EATING ;
          self[i].signal () ;
       }
 }

  initialization_code() {
      for (int i = 0; i < 5; i++)
      state[i] = THINKING;
 }
}
```



## Thread examples: MUTEX

```
/* mutex are only valid within the same process */
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INI
int counter=0;

/* Function C */
void functionC()
{
pthread_mutex_lock( &mutex1 );
counter++
pthread_mutex_unlock( &mutex1 );
}
```



## Mutex example program: mutex1.c

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

void *functionC();
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int counter = 0;

main()
{
int rc1, rc2;
pthread_t thread1, thread2;

/* Create independent threads each of which will execute functionC */

if( (rc1=pthread_create( &thread1, NULL, &functionC, NULL)) )
{
printf("Thread creation failed: %d\n", rc1);
}

if( (rc2=pthread_create( &thread2, NULL, &functionC, NULL)) )
{
printf("Thread creation failed: %d\n", rc2);
}

/* Wait till threads are complete before main continues. Unless we */
/* wait we run the risk of executing an exit which will terminate */
/* the process and all threads before the threads have completed. */

pthread_join( thread1, NULL);
pthread_join( thread2, NULL);

exit(0);
}

void *functionC()
{
pthread_mutex_lock( &mutex1 );
counter++;
printf("Counter value: %d\n",counter);
pthread_mutex_unlock( &mutex1 );
}
```



## Compile mutex1.c and run

Compile: gcc -lpthread mutex1.c

Run: ./a.out

Results:

Counter value: 1
Counter value: 2

# THIS IS ALL ABOUT THREADS!