Dining Philosophers

Five philosophers sit around a table with five forks and spaghetti to eat.

Philosophers think for a while and they want to eat, only spaghetti, for a while.

To eat a philosopher requires two forks, one from the left and one from right.

Assume a philo. can only pick up one fork at a time.

After eating, forks are placed down and philo. goes back to thinking.

Dining Philosophers

Devise an algorithm that will allow philosophers to eat.

Must satisfy mutual exclusion
- no two philos can use the same fork at the same time.

Avoid deadlock and starvation!

Dining Philosophers

First attempt:

- take left fork, then take right fork
- Wrong! Results in deadlock.

Second attempt:

- take left fork, check to see if right is available, if not put left one down.
- Still has race condition and can lead to starvation.
- Change so wait a random time. Will work usually, but want a solution that will always work, guaranteed.

Dining Philosophers

One solution is to protect obtaining forks with a binary semaphore (mutex) - but only one philo can eat at a time rather than two.

Following code uses one semaphore per philo and states of eating, thinking, and hungry. A philo can advance to eating if neither neighbor is eating.

Dining Philosophers

```
#define N 5   /* number of philos */
#define LEFT (i+N-1)%N  /* # of i's left */
#define RIGHT (i+1)%N   /* # of i's right */
#define THINKING 0
#define HUNGRY 1
#define EATING 2

int state[N];            /* keep track of state */
semaphore mutex = 1;
semaphore s[N]; /* semaphore per philo */
```

Dining Philosophers

```
void philosopher(int i)
{
  while (TRUE) {
    think();                /* thinking */
    take_forks(i);  /* get two forks, block */
    eat();                  /* eating */
    put_forks(i);   /* give up forks */
  }
}
```

Dining Philosophers

```
void take_forks(int i)
{
  wait(mutex);
  state[i] = HUNGRY;
  test(i);            /* try getting 2 forks */
  signal(mutex);
  wait(s[i]);    /* block if no forks acquired */
}
```

Dining Philosophers

```
void test(int i)
{
  if ( state[i] == HUNGRY &&
       state[LEFT] != EATING &&
       state[RIGHT] != EATING ) {
          signal(s[i]);
  }
}
```

Dining Philosophers

```
void put_forks(int i)
{
  wait(mutex);
  state[i] = THINKING;
  test(LEFT);
  test(RIGHT);
  signal(mutex);
}
```

Dining Philosophers

Another solution is to have some philos be left handed first.

Others:

- get more forks, learn to eat with one fork. (changes the problem)
- only let four at a time to the table

Readers/Writers

Models access to a database.
Some processes want to read the DB others write it.
Many readers can read at the same time.
But when writing mutual exclusion is needed.

Following code could shut out writers if enough readers.

Readers/Writers

```
semaphore mutex = 1;
semaphore db = 1;
int rc = 0;

void writer()
{ while (TRUE) {
    think_up_data();
    wait(db);
    write_data_base();
    signal(db);
  } }
```

Readers/Writers

```
void reader()
{ while (TRUE) {
    wait(mutex);  rc = rc + 1;
    if ( rc == 1 ) wait(db);
    signal(mutex);
    read_data_base();
    wait(mutex);  rc = rc – 1;
    if ( rc == 0 ) signal(db);
    signal(mutex);
    use_data_read();
} }
```

Sleeping Barber

Barber shop with one barber, one barber chair and N chairs to wait in.

When no customers the barber goes to sleep in barber chair and must be woken when a customer comes in.

When barber is cutting hair new customers take empty seats to wait, or leave if no vacancy.

Program barber and customers so no race condition exists.

Sleeping Barber

The following code uses 3 semaphores.

- customers – number of waiting custs
- barbers – number of barbers (0 or 1)
  that are idle
- mutex

Sleeping Barber

```
#define CHAIRS 5     /* # of chairs */

semaphore customers = 0;
semaphore barbers = 0;
semaphore mutex = 1;

int waiting = 0;
```

Sleeping Barber

```
void barber()
{
  while (TRUE) {
    wait(customers);
    wait(mutex);
    waiting = waiting – 1;
    signal(barbers);
    signal(mutex);
    cut_hair();
  }
}
```

Sleeping Barber

```
void customer()
{  wait(mutex);
    if ( waiting < CHAIRS ) {
      waiting = waiting + 1;
    signal(customers);
    signal(mutex);
    wait(barbers);
    get_haircut();
  } else {
      signal (mutex); }
}
```