The Dining Philosophers Problem

The Dining Philosophers Problem

It is an artificial problem widely used to illustrate the problems linked to resource sharing in concurrent programming

The dining philosophers problem: a first solution

This first solution uses a semaphore to model each fork.
- Taking a fork is then done by executing a operation wait on the semaphore, which suspends the process if the fork is not available.
- Freeing a fork is naturally done with a signal operation.

Solution 1: Semaphore

```
/* Definitions and global initializations */
#define N = ? /* number of philosophers */
semaphore fork[N]; /* semaphores modeling the forks */
philospher(i)
int i;
int j;
{
    for (j=0, j < N, j++)
        fork[j]=1;
    while(true)
    {
        think();
        wait(fork[i]); wait(fork[(i+1)%N]);
        eat();
        signal(fork[i]); signal(fork[(i+1)%N]);
    }
}
```

Each philosopher (0 to N-1) corresponds to a process executing the following procedure, where *i* is the number of the philosopher.

Semaphore Solution - Deadlock

- With the first solution, a deadlock is possible
- Indeed, if each philosopher executes `wait(fork[i])` before any philosopher has executed `wait(fork[(i+1)%N])`, each philosopher is then holding one fork and waiting for the second.
- The problem is that each philosopher must acquire two resources and does this
  1. in two steps,
  2. in an order that can lead to a deadlock, and
  3. without the possibility of the operation acting canceled

To avoid deadlocks, one of these three items has to be eliminated.

The dining philosophers problem: a second solution

To this solution, the order in which the philosopher will pick up his forks is modified

```
/* Philosophe 0 a N-2 */
philospher(i)
int i;
{ while(true)
  { think();
    wait(fork[i]);
    wait(fork[(i+1)%N]);
    eat();
    signal(fork[i]);
    signal(fork[(i+1)%N]);
  }
}
```

```
/* Philosophe N-1*/
philospher(i)
int i;
{ while(true)
  { think();
    wait(fork[(i+1)%N]);
    wait(fork[i]);
    eat();
    signal(fork[(i+1)%N]);
    signal(fork[i]);
  }
}
```

Tannenbaum's Solution: Binary Semaphore

```
system DINING_PHILOSOPHERS

var
me:    semaphore, initially 1;                  /* for mutual exclusion */
s[5]:  semaphore s[5], initially 0;             /* for synchronization */
pflag[5]: (THINK, HUNGRY, EAT), initially THINK;  /* philosopher flag */

 procedure philosopher(i)
   {
     while TRUE do
       {
         THINKING;
         take_chopsticks(i);
         EATING;
         drop_chopsticks(i);
       }
   }
```

Tannenbaum's Solution: Binary Semaphore

```
procedure take_chopsticks(i)
  {
    DOWN(me);                  /* critical section */
    pflag[i] := HUNGRY;
    test[i];
    UP(me);                    /* end critical section */
    DOWN(s[i])                 /* Eat if enabled */
  }

void test(i)              /* Let phil[i] eat, if waiting */
  {
    if ( pflag[i] == HUNGRY
      && pflag[i-1] != EAT
      && pflag[i-1] != EAT)
       then
        {
          pflag[i] := EAT
          UP(s[i])
        }
  }
```

Tannenbaum's Solution: Binary Semaphore

```
void drop_chopsticks(int i)
  {
    DOWN(me);                  /* critical section */
    test[i-1];                 /* Let phil. on left eat if possible */
    test[i+1];                 /* Let phil. on right eat if possible */
    UP(me);                    /* up critical section */
  }
```

http://www.cs.indiana.edu/classes/p415-sjoh/hw/project/dining-philosophers/index.htm

http://www.cs.grinnell.edu/~weinman/courses/CSC213/2012F/labs/philosophers.html

http://stackoverflow.com/questions/18122499/memory-leak-in-java-dining-philosophers-implemented-with-semaphores

The Dining Philosophers Problem

The Dining Philosophers Problem

It is an artificial problem widely used to illustrate the problems linked to resource sharing in concurrent programming

The dining philosophers problem: a first solution

This first solution uses a semaphore to model each fork.
- Taking a fork is then done by executing a operation wait on the semaphore, which suspends the process if the fork is not available.
- Freeing a fork is naturally done with a signal operation.

Solution 1: Semaphore

```
/* Definitions and global initializations */
#define N = ? /* number of philosophers */
semaphore fork[N]; /* semaphores modeling the forks */
philospher(i)
int i;
int j;
{
    for (j=0, j < N, j++)
        fork[j]=1;
    while(true)
    {
        think();
        wait(fork[i]); wait(fork[(i+1)%N]);
        eat();
        signal(fork[i]); signal(fork[(i+1)%N]);
    }
}
```

Each philosopher (0 to N-1) corresponds to a process executing the following procedure, where *i* is the number of the philosopher.

Semaphore Solution - Deadlock

- With the first solution, a deadlock is possible
- Indeed, if each philosopher executes `wait(fork[i])` before any philosopher has executed `wait(fork[(i+1)%N])`, each philosopher is then holding one fork and waiting for the second.
- The problem is that each philosopher must acquire two resources and does this
  1. in two steps,
  2. in an order that can lead to a deadlock, and
  3. without the possibility of the operation acting canceled

To avoid deadlocks, one of these three items has to be eliminated.

The dining philosophers problem: a second solution

To this solution, the order in which the philosopher will pick up his forks is modified

```
/* Philosophe 0 a N-2 */
philospher(i)
int i;
{ while(true)
  { think();
    wait(fork[i]);
    wait(fork[(i+1)%N]);
    eat();
    signal(fork[i]);
    signal(fork[(i+1)%N]);
  }
}
```

```
/* Philosophe N-1*/
philospher(i)
int i;
{ while(true)
  { think();
    wait(fork[(i+1)%N]);
    wait(fork[i]);
    eat();
    signal(fork[(i+1)%N]);
    signal(fork[i]);
  }
}
```

Tannenbaum's Solution: Binary Semaphore

```
system DINING_PHILOSOPHERS

var
me:    semaphore, initially 1;                  /* for mutual exclusion */
s[5]:  semaphore s[5], initially 0;             /* for synchronization */
pflag[5]: (THINK, HUNGRY, EAT), initially THINK;  /* philosopher flag */

 procedure philosopher(i)
   {
     while TRUE do
       {
         THINKING;
         take_chopsticks(i);
         EATING;
         drop_chopsticks(i);
       }
   }
```

Tannenbaum's Solution: Binary Semaphore

```
procedure take_chopsticks(i)
  {
    DOWN(me);                  /* critical section */
    pflag[i] := HUNGRY;
    test[i];
    UP(me);                    /* end critical section */
    DOWN(s[i])                 /* Eat if enabled */
  }

void test(i)              /* Let phil[i] eat, if waiting */
  {
    if ( pflag[i] == HUNGRY
      && pflag[i-1] != EAT
      && pflag[i-1] != EAT)
       then
        {
          pflag[i] := EAT
          UP(s[i])
        }
  }
```

Tannenbaum's Solution: Binary Semaphore

```
void drop_chopsticks(int i)
  {
    DOWN(me);                  /* critical section */
    test[i-1];                 /* Let phil. on left eat if possible */
    test[i+1];                 /* Let phil. on right eat if possible */
    UP(me);                    /* up critical section */
  }
```

http://www.cs.indiana.edu/classes/p415-sjoh/hw/project/dining-philosophers/index.htm

http://www.cs.grinnell.edu/~weinman/courses/CSC213/2012F/labs/philosophers.html

http://stackoverflow.com/questions/18122499/memory-leak-in-java-dining-philosophers-implemented-with-semaphores

The Dining Philosophers Problem

It is an artificial problem widely used to illustrate the problems linked to resource sharing in concurrent programming

The dining philosophers problem: a first solution

This first solution uses a semaphore to model each fork.

- Taking a fork is then done by executing a operation wait on the semaphore, which suspends the process if the fork is not available.
- Freeing a fork is naturally done with a signal operation.

Solution 1: Semaphore

```
/* Definitions and global initializations */
#define N = ? /* number of philosophers */
semaphore fork[N]; /* semaphores modeling
the forks */
int j;
for (j=0, j < N, j++)
     fork[j]=1;
philospher(i)
int i;
{
    while(true)
    {
        think();
        wait(fork[i]); wait(fork[(i+1)%N]);
        eat();
        signal(fork[i]); signal(fork[(i+1)%N]);
    }
}
```

Each philosopher (0 to N-1) corresponds to a process executing the following procedure, where *i* is the number of the philosopher.

Semaphore Solution - Deadlock

- With this first solution, a deadlock is possible.
- Indeed, if each philosopher executes `wait(fork[i])` before any philosopher has executed `wait(fork[(i+1)%N])`, each philosopher is then holding one fork and waiting for the second.
- The problem is that each philosopher must acquire two resources and does this
  1. in two steps,
  2. in an order that can lead to a deadlock, and
  3. without the possibility of the operation being canceled

To avoid deadlocks, one of these three items has to be eliminated.

The dining philosophers problem: a second solution

In this solution, the order in which the philosopher N-1 picks up his forks is modified.

```
/* Philosophes 0 à N-2 */
philospher(i)
int i;
{ while(true)
  { think();
    wait(fork[i]);
    wait(fork[(i+1)%N]);
    eat();
    signal(fork[i]);
    signal(fork[(i+1)%N]);

  }
}
```

```
/* Philosophe N-1*/
philospher(i)
int i;
{ while(true)
  { think();
    wait(fork[(i+1)%N]);
    wait(fork[i]);
    eat();
    signal(fork[(i+1)%N]);
    signal(fork[i]);
  }
}
```

Tannenbaum's Solution: Binary Semaphore

```
system DINING_PHILOSOPHERS

VAR
me:     semaphore, initially 1;                          /* for mutual exclusion */
s[5]:   semaphore s[5], initially 0;                     /* for synchronization */
pflag[5]: {THINK, HUNGRY, EAT}, initially THINK;         /* philosopher flag */
```

```
procedure philosopher(i)
   {
     while TRUE do
      {
        THINKING;
        take_chopsticks(i);
        EATING;
        drop_chopsticks(i);
      }
   }
```

Tannenbaum's Solution: Binary Semaphore

```
procedure take_chopsticks(i)
   {
     DOWN(me);                    /* critical section */
     pflag[i] := HUNGRY;
     test[i];
     UP(me);                      /* end critical section */
     DOWN(s[i])                   /* Eat if enabled */
    }

void test(i)                 /* Let phil[i] eat, if waiting */
   {
     if ( pflag[i] == HUNGRY
       && pflag[i-1] != EAT
       && pflag[i+1] != EAT)
        then
         {
           pflag[i] := EAT;
           UP(s[i])
          }
     }
```

Tannenbaum's Solution: Binary Semaphore

```
void drop_chopsticks(int i)
  {
    DOWN(me);                    /* critical section */
    test(i-1);                   /* Let phil. on left eat if possible */
    test(i+1);                   /* Let phil. on rght eat if possible */
    UP(me);                      /* up critical section */
   }
```

http://www.cs.indiana.edu/classes/p415-sjoh/hw/project/dining-philosophers/index.htm

http://www.cs.grinnell.edu/~weinman/courses/CSC213/2012F/labs/philosophers.html

http://stackoverflow.com/questions/18122499/memory-leak-in-java-dining-philosophers-implemented-with-semaphores