Strategy 4 - Deadlock Prevention – The Banker's Algorithm

- The main algorithms for preventing deadlock are based upon the concept of *safe and unsafe states*.
- Such algorithms avoid deadlock by preventing a circular list of dependencies forming between a set a processes and the resources they are attempting to acquire. (Remember that this was one of the 4 'Coffman' conditions required for a system to be deadlock).
- Preventing this condition arising thus prevents the possibility of deadlock occurring. Stated simply, you deny a process's request to acquire a resource if it would lead to a circular depending list forming in the first case.
- In essence the system always tries to maintain itself in a safe state where by there are always enough free resources to satisfy at least one process, i.e. there is always at least one process that can proceed and is not blocked.
- Dijkstra came up with a scheduling algorithm/solution known as the *Bankers algorithm* because it was modeled upon the way that a small town banker might deal with a group of customers to whom he has granted credit or overdraft arrangement. Such a scheme can be employed by any operating system wishing to prevent deadlock occurring. Let's see how it works.

Strategy 4 - The Banker's Algorithm – an Explanation

- Imagine that a banker (in this analogy our operating system) has several customers (i.e. processes) which need to borrow certain sums of money (i.e. resources) which they intend to replay (i.e. they will return the resource)
- For example, let's suppose the banker has 4 customers, we'll call them A, B, C and D, each of which have agreed, in advance with the banker, that they can borrow a certain amount of money in the future i.e. they have agreed in principle to a loan or overdraft.

  (This is analogous to our processes negotiating in advance with the operating system that they intend to borrow several resources in the future).

- Let's suppose that the banker has agreed in principle that

  - A can borrow $6000
  - B can borrow $5000
  - C can borrow $4000 and
  - D can borrow the most at $7000,

- That is, the banker has agreed in principle to loans totaling $22000

Strategy 4 - The Banker's Algorithm – an Explanation (cont…)

- The problem here (which is an important point in the discussion on deadlock prevention) is the fact that the banker doesn't actually have $22000 to lend. At least not all at once !!.
- In fact he only has $10000, but this is still more than sufficient than the maximum amount he has agreed to lend to any one of his customer (Customer D at $7000)
- In effect the banker is gambling/relying on the fact that not all of the customers will want to borrow all their entitlement all at the same time.
- In other words he is betting that the loans will be phased at different times and that some of them may actually pay back some or all of their loans before he runs out of money to lend.
- In our computing analogy, this is equivalent to saying that an operating system has 10 resources, but that processes A, B, C and D might actually want 22 if they all request those resources simultaneously. In other words there is the potential for a deadlock, which may or may not happen depending upon the order and frequency with whiccj the banker lends and received money.
- However the banker and the operating system still have enough money/resources in reserve at all times to meet each individual customers/processes request for its maximum remaining allocation of money/resources.
- Let's see what happens

Strategy 4 - The Banker's Algorithm – an Explanation (cont...)

- When the banker opens his bank on Monday morning, none of the customers have borrowed any money.
- This initial state of play is shown in Fig (a) below and at this point at least, the banker can satisfy any single request made by any one customer for their full loan entitlement.
- However, some of his customers may arrive at various times during the day and request some or perhaps their entire agreed loan (this is equivalent to several processes requesting some or all of their agreed resources from the operating system).
- Let's move on and consider Fig (b). Here, at say 3.30pm the banker has given $1000 to Customers A and B, $2000 to C and $4000 to D. Thus he has loaned an amount totaling $8000 and now has just $2000 in reserve.
- The important point to note here is that there is still enough in reserve to satisfy at least one of his customers if they request the remainder of their full entitlement; in this case customer C who can borrow another $2000.
- This state of affairs is referred to as a safe state, because at least one of his customers can proceed with their full loan i.e. they cannot be deadlocked because the banker has enough resources to allow at least one of his customers to borrow money.

	Has	Max
A	0	6
B	0	5
C	0	4
D	0	7

Free: 10

(a)

	Has	Max
A	1	6
B	1	5
C	2	4
D	4	7

Free: 2

(b)

	Has	Max
A	1	6
B	2	5
C	2	4
D	4	7

Free: 1

(c)

Strategy 4 - The Banker's Algorithm – an Explanation (cont…)

- Let's suppose now that at 3.40pm, customer B, requests a further $1000 of his agreed loan, as shown in Fig (c). This would mean that the banker now only has $1000 in reserve.
- This is an unsafe state, because the banker does not have enough in reserve to satisfy any one of his customers if they call upon their full loan entitlement and a deadlock situation could arise.
- However, it does not mean that a deadlock will arise, it's just that the potential exists and to avoid it, the banker would temporarily refuse the loan of the £1k to customer B in Fig (c) until some other customer had paid back sufficient of their loan such that granting B's request would still leave the system in a safe state.
- The idea then is that a safe state is one where an operating system has sufficient resources in reserve to satisfy the full amount of outstanding resource requests that could be made by at least one process.
- If a request is made that would lead the operating system into an unsafe state, then that request is temporarily refused and the process will be put on hold.

	Has	Max
A	0	6
B	0	5
C	0	4
D	0	7

Free: 10

(a)

	Has	Max
A	1	6
B	1	5
C	2	4
D	4	7

Free: 2

(b)

	Has	Max
A	1	6
B	2	5
C	2	4
D	4	7

Free: 1

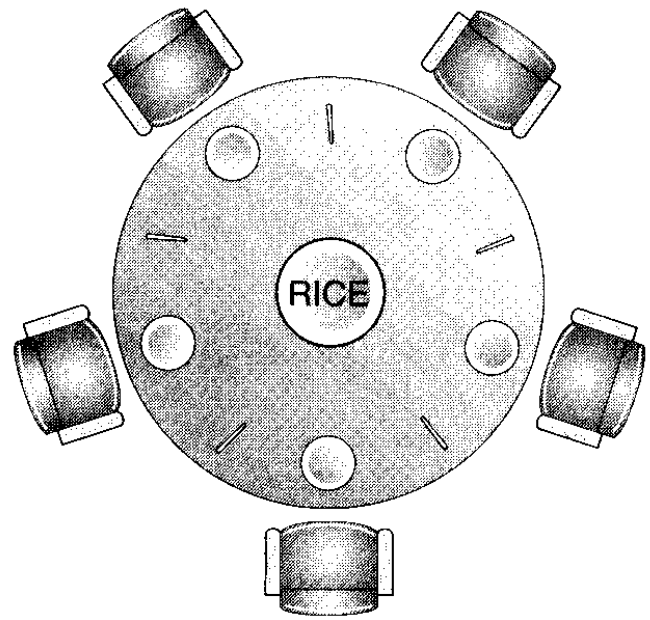
(c)

Limitations of the Banker's Algorithm

- The main problem or limitation of this algorithm, as you have probably identified yourself, is the fact that processes have to declare in advance the number of each type of resource they are intending to use.
- If they cannot do this, then deadlock prevention using this scheme cannot be assured.
- Such a scheme works well in batch programming, but in interactive or real-time systems, where processes are created in response to users logging in or events triggering the creation of new threads (which may request resources) it cannot be employed safely and the risk of deadlock will continue.

Classical Deadlock Problems –

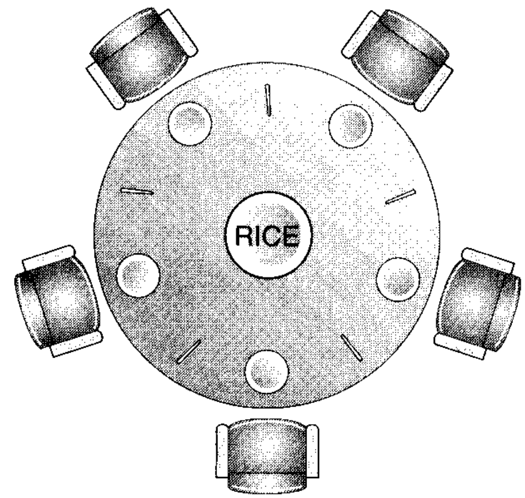
The Dinning Philosophers



- The picture opposite illustrates a classical deadlock and starvation problem known commonly as the Dinning Philosophers problem and was first proposed by Dijkstra as a vehicle for operating systems designers to test their deadlock detection and recovery scheme against
- It also serve a dual purpose in that it gives students of operating systems and concurrent programming something of a problem to keep them awake at night in their attempts to understand its solution.

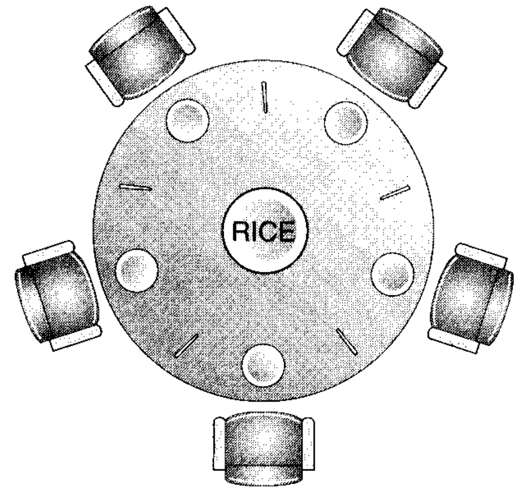
Deadlock and Starvation

The Dinning Philosophers – An Explanation



- The original problem concerned five philosophers who sit around a circular table. Each philosopher has their own bowl and access to two chopsticks which are also shared with each of the philosopher's neighbours. In fact the chopsticks are the classical resources that each philosopher fights over.
- The life of a philosopher consists of alternate bouts of eating, for which he needs two chopsticks, and thinking which requires none. The problem here is one of both deadlock and starvation.
- Imagine all philosophers decide to eat at the same time, if they all pick up their left chopstick first, followed by the right, then deadlock will occur, ditto if they pick of their right first followed by their left.
- We could modify the approach so that a philosopher picks up their left chopstick and then checks if the right is available. If it is not, he puts down his left chopstick and tries again later.
- Unfortunately, this solution can lead to starvation as we saw with voluntary relinquishment of a resource, whereby if a philosophers gives up a chopstick he/she may never get it back again.

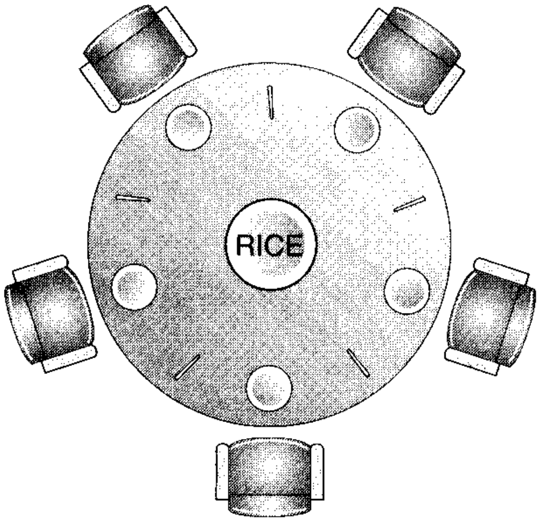
The Dinning Philosophers – A Solution



- The solution always presented in books is rather terse and cryptic and even though it is short, it is quite difficult to follow.
- An example of it, implemented as a class can be found in the rt.cpp library header file.
- Look for the class CDinningPhilosophers, while a working solution/implementation of it can be found in Q12 of the tutorial questions.
- Even then, the solution is not very good and is actually quite unrealistic since it doesn't really involve philosophers attempting to acquire the resources and if they are busy releasing the ones they already possess (as they would in the real world), but rather involves a philosopher testing the state of his neighbouring philosopher to see whether they are eating or not and hence deducing whether or not the chopsticks are free, which isn't really the point.
- It's a bit like process A checking what process B is doing and hence deducing whether B is using the resource that A wants.
- Thus the whole solution isn't even a generic solution since it is a specific solution to a specific problem, i.e. the dinning philosopher problem
- A more generic solution is actually a lot simpler and can easily be tailored to solving the dinning philosophers problem as well as other problems and makes use of a specific Win32 call (which other operating systems will have in some form or another) called the WAIT_FOR_MULTIPLE_OBJECTS()

Deadlock and Starvation

The Dinning Philosophers – A Solution (cont…)



- The WAIT_FOR_MULTIPLE_OBJECTS system call in the Win32 kernel simply allows a process/thread to request multiple resources simultaneously.
- The kernel will ensure that that none of the resources that the process/thread is requesting will be given to it until is can simultaneously acquire them all.
- In effect this behaves much like the deadlock prevention scheme discussed earlier, where multiple resources were treated as one resource. Either a process/thread has them all, or it has none of them.
- However this scheme has the added flexibility that other processes/thread can request a sub-set of those resources and thus does not suffer from the limitations of that scheme.
- It does not however eliminate the possibility of starvation, since even if the call manages to locate a subset of the requested resource as being available, it cannot acquire them until they are all available and thus the possibility exists that another process/thread will acquire the available ones first.
- If this happens frequently enough, then a process/thread may never find all its requested resource available and thus 'starve'

The Dinning Philosophers – A Solution

- To demonstrate the use of the WAIT_FOR_MULTIPLE_OBJECTS call, imagine we have two separate resources, a scanner and a CD Writer with their corresponding two semaphores to protect them, as shown below.

  - CMutex Scanner(……) ;
  - CMutex CDWriter(……) ;

- Now imagine a process wants to acquire both of these resources without the possibility of deadlock. It obviously has to wait for both of them to become free and thus we cannot acquire one then the other sequentially, we have to acquire them both simultaneously. We could achieve this using the WAIT_FOR_MULTIPLE_OBJECTS call as shown below.
- First we have to create an array of ‘HANDLES’ (these are for all intents and purposes identifies for the mutex’s) to the resources we want to acquire; in this case 2.

```
HANDLE MyResourceHandles[2] = {
        ScannerMutex.GetHandle(),                   // get handle for each mutex
        CDWriterMutex.GetHandle()
} ;
```

Deadlock and Starvation

The Dinning Philosophers – A Solution

- Having got the two handles to the two resources we could now attempt to acquire them thus

```
WAIT_FOR_MULTIPLE_OBJECTS(
                    2,                  // number of resources to acquire
                    MyResourceHandles,  // array of handles to those resources
                    INFINITE            // try for this length of time to acquire them or give up
) ;
```

- Here, we give the function the array of handles to the resources we require and then tell it to acquire them for us. We have specified a time period of INFINITE meaning do not return until you have got them.
- Once the function returns, we know that we have sole access to the two resources and that a Wait() operation will have been performed on the resources so that nobody else can acquire them.
- We still of course have to Signal() the resources in the usual manner when we have finished with them.
- Armed with this, we could thus solve the dinning philosophers problem as shown below

Deadlock and Starvation


```
#define     NUMBER_OF_PHILOS      5                       // start off with 5 philosophers
CMutex      *Chopsticks[NUMBER_OF_PHILOS] ;               // create an array of mutex's, one for each philosopher

//    This function returns the index of the chopstick mutex to the left of the current philosopher
int   LeftChopstick(int i)     { return ((i + NUMBER_OF_PHILOS - 1) % NUMBER_OF_PHILOS) ;   }

//    This function returns the index of the chopstick mutex to the left of the current philosopher
int   RightChopstick(int i)    {return ((i + 1) % NUMBER_OF_PHILOS) ;      }

//    A thread to represent a philosopher
UINT _ _stdcall Philosopher(void *args)
{
      int         MySeatNumber = *(int *)(args) ;                 // figure out which philosopher I am

      // Now get the indexes into the array for my left and right chopsticks semaphores
      int MyLeftChopstickNumber = LeftChopstick(MySeatNumber) ;
      int MyRightChopstickNumber = RightChopstick(MySeatNumber) ;

      //          Now get the Win32 'Handles' for those left and right chopstick semaphores and put them into an array
      //          this is needed for the Wait_for_multiple_objects call below

      HANDLE MyChopstickHandles[2] = {  Chopsticks[MyLeftChopstickNumber]->GetHandle(),   Chopsticks[MyRightChopstickNumber]->GetHandle() } ;

      for(int i = 0; i < 10; i++)             {
                  SLEEP(300 + (MySeatNumber * 300)) ;             // simulate thinking, use a different time delay for each philosopher
                  // Now simulate Eating by acquiring chopsticks to left and right of me
                  WAIT_FOR_MULTIPLE_OBJECTS(2, MyChopstickHandles, INFINITE) ;
                  printf("%d ", MySeatNumber) ;                   // show philosopher is eating by printing his number
                  SLEEP(300 + (MySeatNumber * 10)) ;              // simulate eating, this is a different time delay for each philosopher

                  Chopsticks[MyLeftChopstickNumber]->Signal() ;   // put down left chopstick, i.e. release the resource after eating
                  Chopsticks[MyRightChopstickNumber]->Signal() ;  // put down right chopstick, i.e. release the resource after eating
      }
      return 0 ;
}
```

Deadlock and Starvation


```
int main()
{
	int SeatNumber[NUMBER_OF_PHILOS];				// create an array of seat numbers for the philosphers
	CThread *thePhilosophers[NUMBER_OF_PHILOS] ;			// create an array of pointers to philosopher threads

	for(int i = 0; i < NUMBER_OF_PHILOS; i ++)		{
			SeatNumber[i] = i ;				// initialise philosopher seat numbers
			char buff[80] ;
			sprintf(buff, "Chopstick%d", i) ;		// generate the name for the mutex based on a number
			Chopsticks[i] = new CMutex(buff) ;				// create the mutex's using generated name
			thePhilosophers[i] = new CThread(Philosopher, ACTIVE, &SeatNumber[i]) ;	// create the philosopher threads
	}

	for(i = 0; i < NUMBER_OF_PHILOS; i ++)		{	// wait for philosophers to terminate
			thePhilosophers[i]->WaitForThread() ;
			delete thePhilosphers[i] ;			// delete thread object
	}

	for(i = 0; i < NUMBER_OF_PHILOS; i ++)			// delete the chopsticks at the end
			delete Chopsticks[i];

	return 0 ;
}
```

- This program can be found in Question 12 of the Tutorial Sheets (Don't worry you don't have to memorize it for an exam)