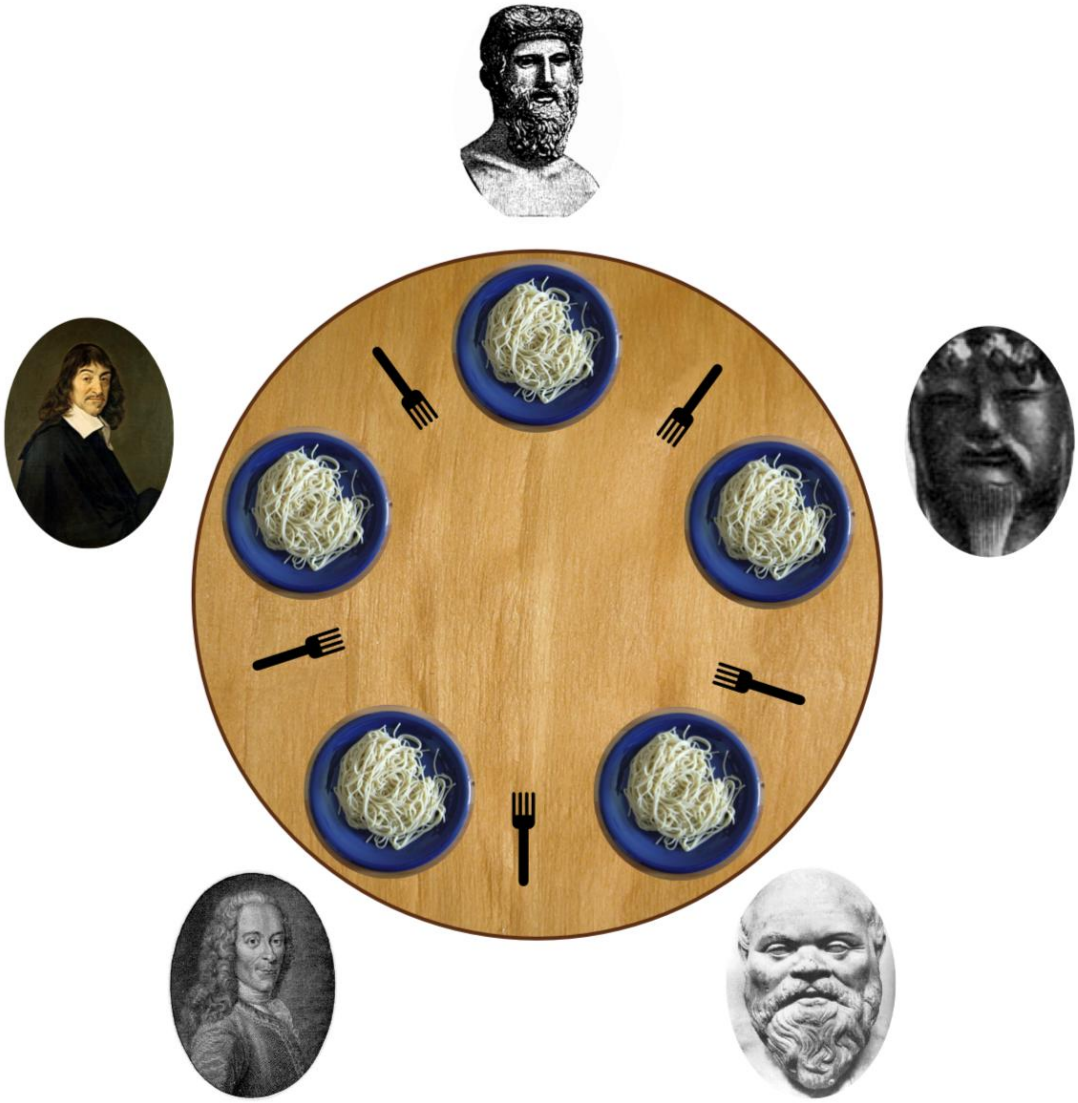
DINING PHILOSOPHERS



BY M.V GANESH RAJU, PROGRAMMING, NETWORKING AND CYBER SECURITY ENTHUSIAST

INTRODUCTION

In computer science, the dining philosophers problem is an example problem often used in concurrent algorithm design to illustrate synchronization issues and techniques for resolving them.

It was originally formulated in 1965 by Edsger Dijkstra as a student exam exercise, presented in terms of computers competing for access to tape drive peripherals. Soon after, Tony Hoare gave the problem its present formulation.

THE PROBLEM STATEMENT

Five silent philosophers sit at a round table with bowls of spaghetti. Forks are placed between each pair of adjacent philosophers.

Each philosopher must alternately think and eat. However, a philosopher can only eat spaghetti when they have both left and right forks. Each fork can be held by only one philosopher and so a philosopher can use the fork only if it is not being used by another philosopher. After an individual philosopher finishes eating, they need to put down both forks so that the forks become available to others. A philosopher can take the fork on their

right or the one on their left as they become available, but cannot start eating before getting both forks.

Eating is not limited by the remaining amounts of spaghetti or stomach space; an infinite supply and an infinite demand are assumed.

The problem is how to design a discipline of behavior (a concurrent algorithm) such that no philosopher will starve; i.e., each can forever continue to alternate between eating and thinking, assuming that no philosopher can know when others may want to eat or think.

Of course, the best thing is to go off and try to solve this problem on our own by exploring various protocols philosophers might use to acquire chopsticks. We are likely to quickly encounter the same deadlock and livelock scenarios we have in the mutual exclusion problem, but we will quickly see in this case that mutual exclusion is too primitive a synchronization mechanism for solving this problem.

In his textbook Modern Operating Systems (Prentice-Hall, 1992) Tannenbaum gives the pseudo-code for a solution to the dining philosophers problem that is shown below. Similar solutions are found in most operating systems textbooks. All of them are renditions of the original

treatment by Dijkstra, which motivated the semaphore mechanism he was introducing in his original article. A \emph{semaphore} is an integer or boolean value, S, with two associated atomic operations:

DOWN(S) wait until S > 0, then decrement S;
UP(S) increment S

In time-sharing systems, "waiting" is implemented by the operating system, which may put processes on a wait-list for later execution. In hardware, "waiting" may be accomplished by busy-waiting or by some form of explicit signaling, such as token passing.

TANNENBAUM'S SOLUTION

This solution uses only Boolean semaphores. There is one global semaphore to provide mutual exclusion for execution of critical protocols. There is one semaphore for each chopstick. In addition, a local two-phase prioritization scheme is used, under which philosophers defer to their neighbors who have declared themselves "hungry." All arithmetic is modulo 5.

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 */
```

As before, each philosopher is an endless cycle of thinking and eating.

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

The take_chopsticks procedure involves checking the status of neighboring philosophers and then declaring one's own intention to eat. This is a two-phase protocol; first declaring the status HUNGRY, then going on to EAT.

```
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])
         }
    }
```

Once a philosopher finishes eating, all that remains is to relinquish the resources---its two chopsticks---and thereby release waiting neighbors.

```
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 */
```

```
}
```

The protocol is fairly elaborate, and Tannenbaum's presentation is made more subtle by its coding style.

A SIMPLER SOLUTION

Other authors, including Dijkstra, have posed simpler solutions to the dining philosopher problem than that proposed by Tannenbaum (depending on one's notion of "simplicity," of course). One such solution is to restrict the number of philosophers allowed access to the table. If there are N chopsticks but only N-1 philosophers allowed to compete for them, at least one will succeed, even if they follow a rigid sequential protocol to acquire their chopsticks.

This solution is implemented with an integer semaphore, initialized to N-1.

Both this and Tannenbaum's solutions avoid deadlock, a situation in which all of the philosophers have grabbed one chopstick and are deterministically waiting for the other, so that there is no hope of recovery.

However, they may still permit starvation, a scenario in which at least one hungry philosopher never gets to eat.

Starvation occurs when the asynchronous semantics may allow an individual to eat repeatedly, thus keeping another from getting a chopstick.

The starving philosopher runs, perhaps, but doesn't make progress. The observation of this fact leads to some further refinement of what fairness means. Under some notions of fairness the solutions given above can be said to be correct.

METHODOLOGY TO IMPLEMENT DINING PHILOSOPHERS USING SEMAPHORES

1. Start.
2. Declare an array of 5 of type sem_t to store the values of the five semaphores.
3. Declare i,j,n as integer variables.
4. Set n=5 as there are 5 semaphores.
5. Declare an array of 5 as pid_t data type to store 5 Process IDs.
6. Declare an array of 5 as char to store the 5 semaphore names.
7. for i=0 to n
   a. Store the Process ID of the current process in an array.
   b. Create a new semaphore and store it in an array.
   c. Print an error message if the creation of the semaphore fails.
   end for
8. for i=0 to n
   a.Create the child processes and if a new child process is created, break.
   end for
9. if i=5
   a.for i=0 to n

i. Wait for the child process to finish executing after which that control switches back to parent.

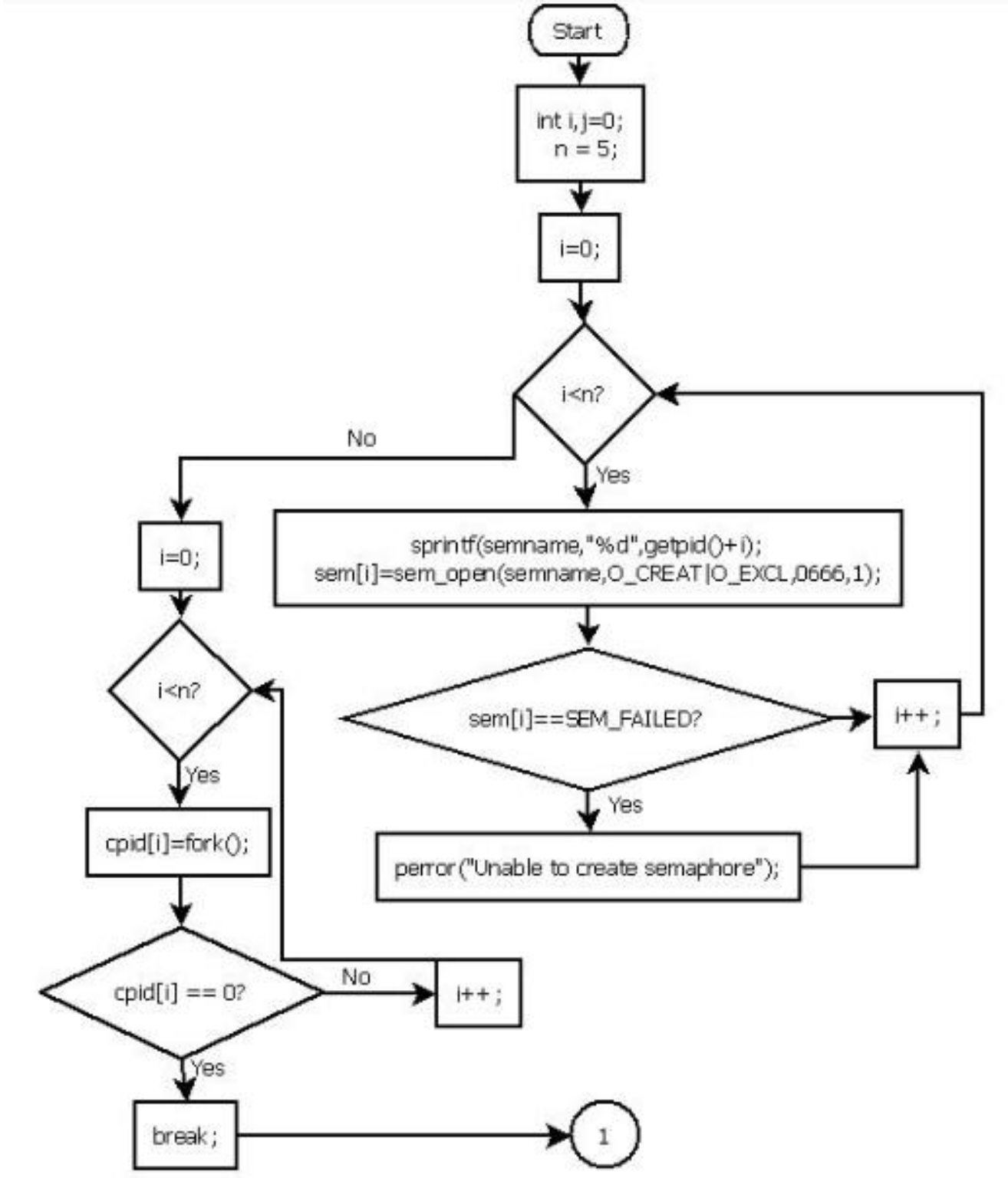
b. for i=0 to n
   i. Close the named semaphore allowing any resources that the system has allocated to be freed.
   ii. Store the process ID in a variable semname
   iii. Remove the process from the system only when all the processes that have it open are exited.

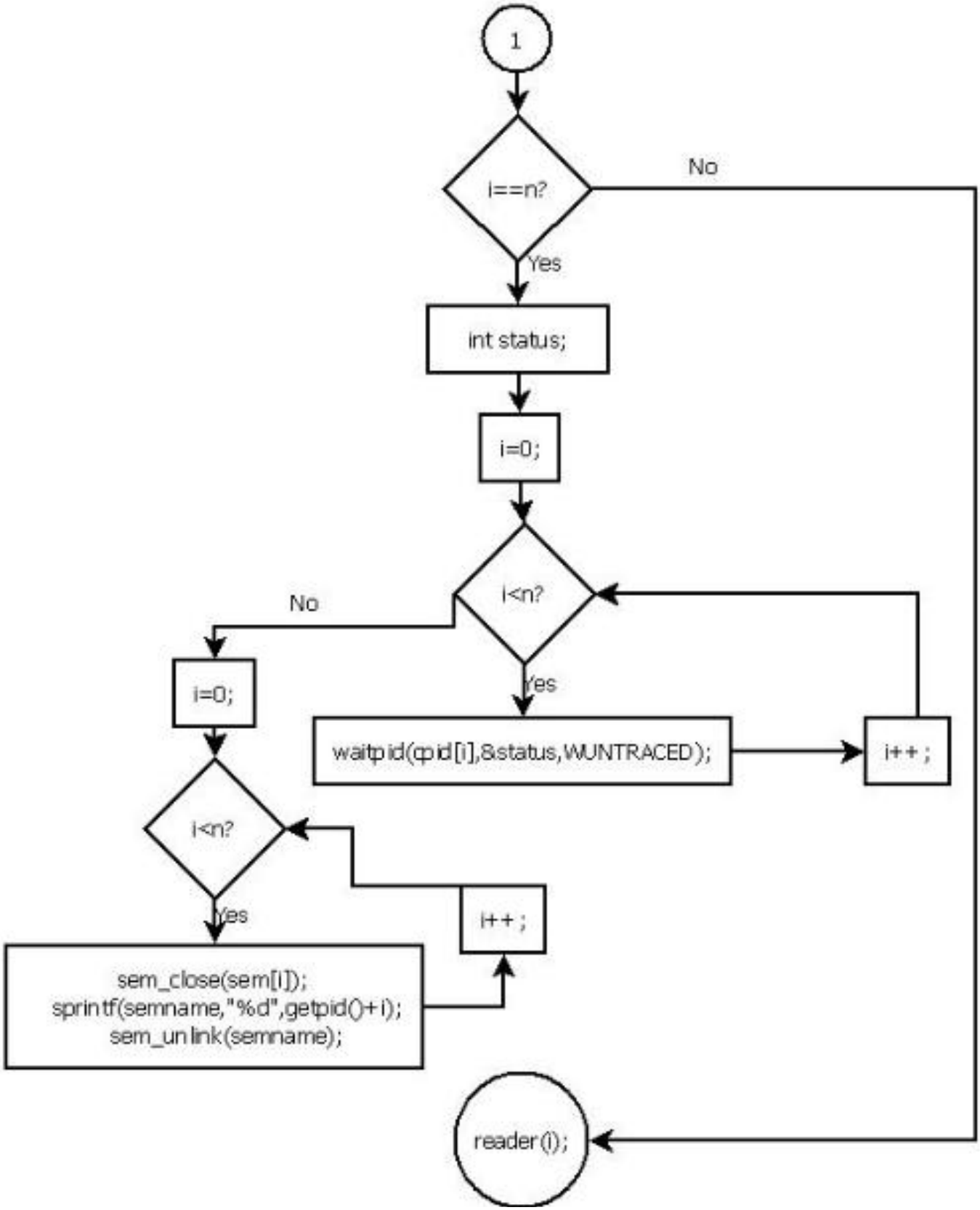
else

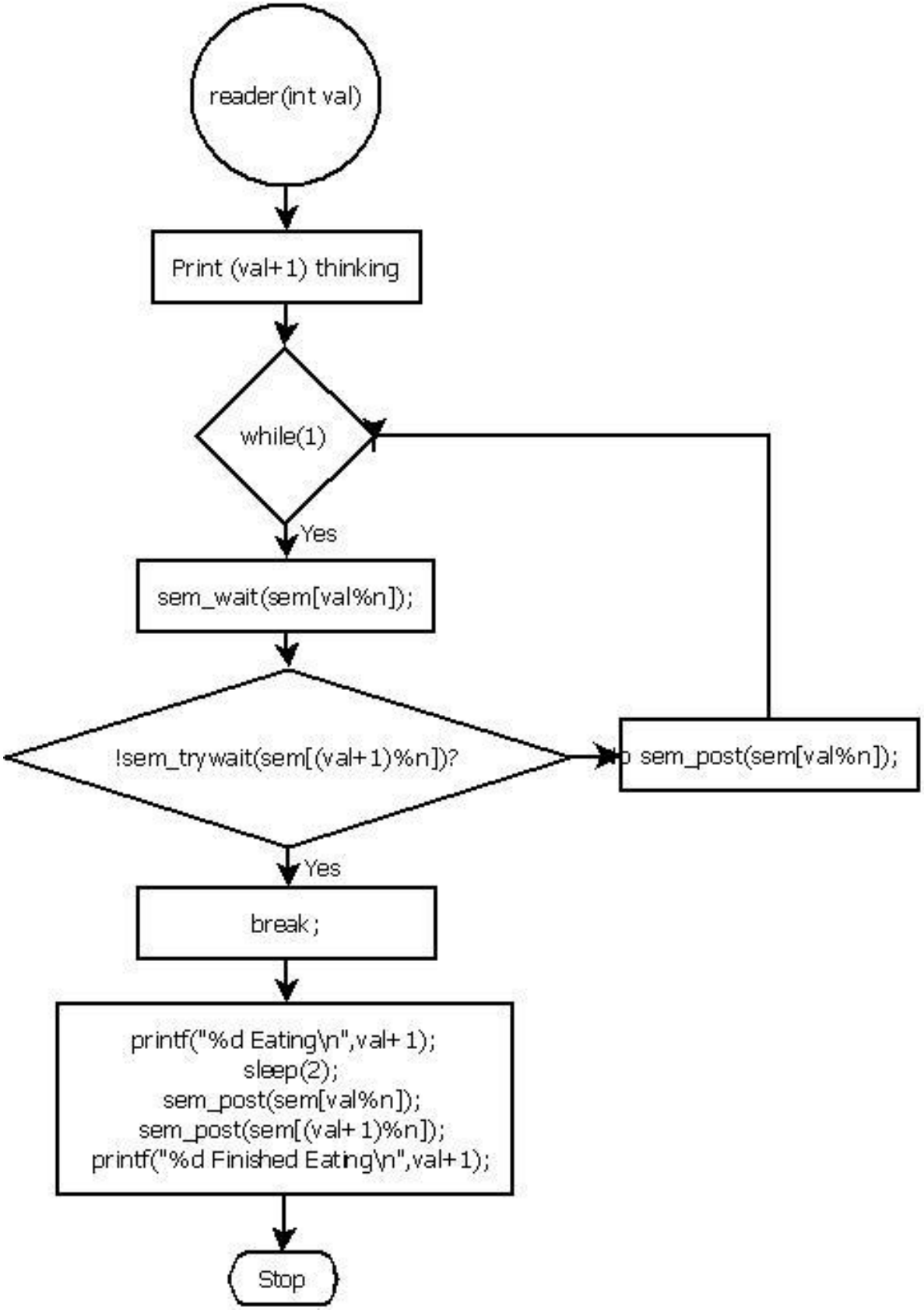
a. Goto 10

10. Print "i" is thinking.
11. while(1)
    a. Lock the semaphore by decrementing the semaphore value.
    b. If the decrement cannot be performed immediately, Goto 12 else unlock the semaphore (i.e. increment it).
12. Print "i+1" is eating
13. Delay the program execution for 2 seconds.
14. Unlock the semaphore i%5
15. Unlock the semaphore (i+1)%5
16. Print "i+1" Finished eating
17. Repeat all the steps until all the philosophers have thought and ate at least once.

FLOWCHART:


Start
int i,j=0;
n = 5;
i=0;
i<n?
No
Yes
i=0;
sprintf(semname,"%d",getpid()+i);
sem[i]=sem_open(semname,O_CREAT|O_EXCL,0666,1);
i<n?
sem[i]==SEM_FAILED?
i++;
Yes
Yes
cpid[i]=fork();
perror("Unable to create semaphore");
cpid[i] == 0?
No
i++;
Yes
break;
1


1
i==n?
No
Yes
int status;
i=0;
i<n?
No
Yes
i=0;
waitpid(cpid[i],&status,WUNTRACED);
i++;
i<n?
Yes
i++;
sem_close(sem[i]);
sprintf(semname,"%d",getpid()+i);
sem_unlink(semname);
reader(i);


reader(int val)
Print (val+1) thinking
while(1)
Yes
sem_wait(sem[val%n]);
!sem_trywait(sem[(val+1)%n])?
sem_post(sem[val%n]);
Yes
break;
printf("%d Eating\n",val+1);
sleep(2);
sem_post(sem[val%n]);
sem_post(sem[(val+1)%n]);
printf("%d Finished Eating\n",val+1);
Stop

CODING:-

```
#include<stdio.h>
#include<fcntl.h>
#include<semaphore.h>
#include<sys/wait.h>
#include<pthread.h>
#include<stdlib.h>

sem_t *sem[20];

/*sem_t type is defined by the header
<semaphore.h> */

int n;
int main()
{
  pid_t cpid[5];

/* pid_t data type is a signed integer type capable
of representing a Process ID */

  char semname[5];
  int i,j=0;
  n = 5;
  for(i=0;i<n;i++)
  {
   sprintf(semname,"%d",getpid()+i);
```

/* The getpid function returns the Process ID of the current process */
/* The C library function sprintf() is used to store formatted data as a string.
 Syntax:
  int sprintf(char *string, const char *float, …, %d we get an integer output */

```
sem[i]=sem_open(semname,O_CREAT|O_EXCL,0666,1);
```

/*The syntax of sem_open() is sem_t *sem_open(const char *sem_name, int oflags,….)
 Here, sem_name is the name of the semaphore that we want to create or access.
 O_CREAT and O_EXCL are flags that affect how the function creates a new semaphore.
When creating a new semaphore, we can set oflags(used argumentally for semaphore creation) to O_CREAT or (O_CREAT|O_EXCEL)
O_CREAT :
 It creates a new named semaphore if we set this bit provide mode and value argument to sem_open()
O_EXCEL:
 When creating a new named semaphore, O_EXCEL causes sem_open() to fail if a semaphore with sem_name already exists.

Without O-EXCL, sem_open() attaches to an existing semaphore or creates a new one if sem_name doesn't exist.
*/

```
   if(sem[i]==SEM_FAILED)
    perror("Unable to create semaphore");
```

/* The C library function void perror(const char *str) prints a descriptive error to stderr. */

```
  }
  for(i=0;i<n;i++)
  {
   cpid[i]=fork();
```

/* fork() is used to create a child process. It takes no arguments and returns a process ID. The purpose of fork() is to create a new process which becomes the child process of the caller.
If fork() < 0 :- It implies that it returns a negative value unsuccessful creation of the child process.
If fork() = 0 :- It implies it returns 0 to the newly created child process.
If fork() > 0 :- It means returns a positive value, the process ID of the child process, to the parent. Both parent and child have identical but separate address spaces. The returned process ID is of type pid_t defined in sys/types.h. Normally, the process

ID is an integer. Moreover, a process can use function getpid() to retrieve the process ID assigned to this process.
*/

```
   if(cpid[i]==0)
    break;
  }
  if(i==n)
  {
    int status;
   for(i=0;i<n;i++)
    waitpid(cpid[i],&status,WUNTRACED);
```

/*waitpid is a function which waits for the child process to finish executing after that control switches back to parent.
WUNTRACED (in stdlib.h):
It's function is that, the status of any child processes specified by pid that are stopped, and whose status has not yet been reported since they stopped, shall also be reported to the requesting process.*/

```
   for(i=0;i<n;i++)
   {
    sem_close(sem[i]);
```

```
/* sem_close() closes the named semaphore which
is in this case sem, allowing any resources that the
system has allocated to the calling process for this
semaphore to be freed. If we don't use
sem_close(), then the semaphore still remains in
the system even if the process exists */

   sprintf(semname,"%d",getpid()+i);
   sem_unlink(semname);

/* sem_unlink: will be removed from the system
only when reference count reaches 0 (i.e all
processes that have it open call sem_close or are
exited.)*/

   }
  }
else
reader(i);
}
int reader(int val)
{
  printf("%d Thinking\n",val+1);
  while(1)
  {
    sem_wait(sem[val%n]);

/* sem_wait() decrements (locks) the semaphore
pointed to by sem. If semaphore's value is > 0,
```

```
the decrement proceeds and function returns. If
semaphore's value is = 0, call blocks until either
it's possible to perform the decrement( s value >
0) or signal or handler interrupts the call */

   if(!sem_trywait(sem[(val+1)%n]))
    break;

/* sem_trywait() is just the same as sem_wait(), it
decrements can't be immediately performed call
returns an error( instead of blocking) */

   else
    sem_post(sem[val%n]);
  }
  printf("%d Eating\n",val+1);
  sleep(2);
/* sleep function delays the program execution for
a given number of seconds. For example, sleep(5);
*/

   sem_post(sem[val%n]);
  sem_post(sem[(val+1)%n]);
  printf("%d Finished Eating\n",val+1);
}

/* sem_post() increments(unlocks) the semaphore
pointed to by sem. If the semaphore's value
consequently becomes greater than zero, then
```

another process or thread blocked in a sem_wait(3) call will be woken up and lock the semaphore. It returns 0 on success. */

Output:

```
1210315127@CSELinx:~/OS$ cc DiningPhilosopher.c -pthread
1210315127@CSELinx:~/OS$ ./a.out
4 Thinking
4 Eating
3 Thinking
5 Thinking
2 Thinking
2 Eating
1 Thinking
4 Finished Eating
5 Eating
2 Finished Eating
3 Eating
5 Finished Eating
1 Eating
3 Finished Eating
1 Finished Eating
```