Interprocess Communication

Interprocess Communication

- Consider shell pipeline
  - ***cat*** **chapter1 chapter2 chapter3** **|** ***grep*** **tree**
  - 2 processes
  - Information sharing
  - Order of execution

Interprocess Communication

- Processes within a system may be **independent** or **cooperating.**
- Cooperating process can affect or be affected by other processes.
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes require a mechanism to exchange data and information.

IPC issues

1. How one process passes information to another ?
2. How to make sure that two or more processes do not get into each other's way when engaging in critical (shared resource)activities?
3. How to do proper sequencing when dependencies are present?

- Ans 1: easy for threads, for processes different approaches (e.g., message passing, shared memory)
- Ans 2 and Ans 3: same problems and same solutions apply for threads and processes
  - **Mutual exclusion & Synchronization**

Spooling Example: Correct

Process 1

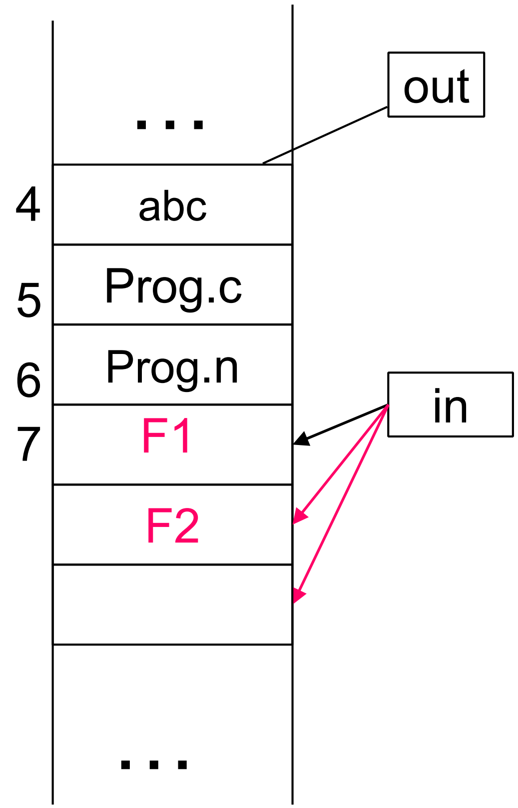
int next_free;

1 next_free = in;

2 Stores F1 into next_free;

3 in=next_free+1

Shared memory



Process 2

int next_free;

4 next_free = in

5 Stores F2 into next_free;

6 in=next_free+1

- Shared memory by multiple processes using a shared variable **in**.
- **Why It's Correct:** Processes are using their local copies (next_free) of the shared in variable.

Spooling Example: Races

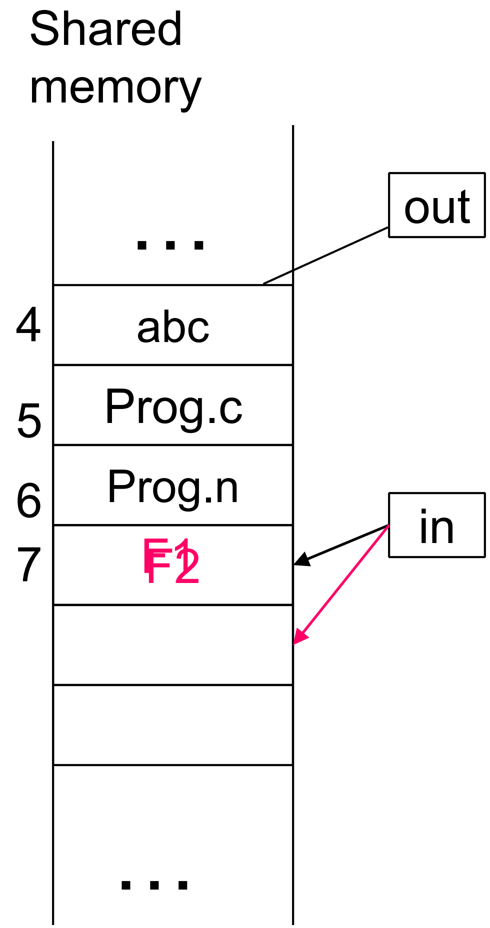
Process 1

int next_free;

1 next_free = in;

3 Stores F1 into next_free;

4 in=next_free+1



Process 2

int next_free;

2 next_free = in
/* value: 7 */

5 Stores F2 into next_free

6 in=next_free+1

Better Coding?

- In previous code

```
for(;;){
    int next_free = in;
    slot[next_free] = file;
    in = next_free+1;
}
```

- What if we use one line of code?

```
for(;;){
    slot[in++] = file
}
```

When Can process Be switched?

- After each machine instruction!
- `in++` is a C/C++ statement, translated into three machine instructions:
  - load mem, R
  - inc R
  - store R, mem
- Interrupt (and hence process swichting) can happen in between.

Race condition

- Two or more processes are reading or writing some shared data and the final result depends on who runs precisely when
- Very hard to Debug

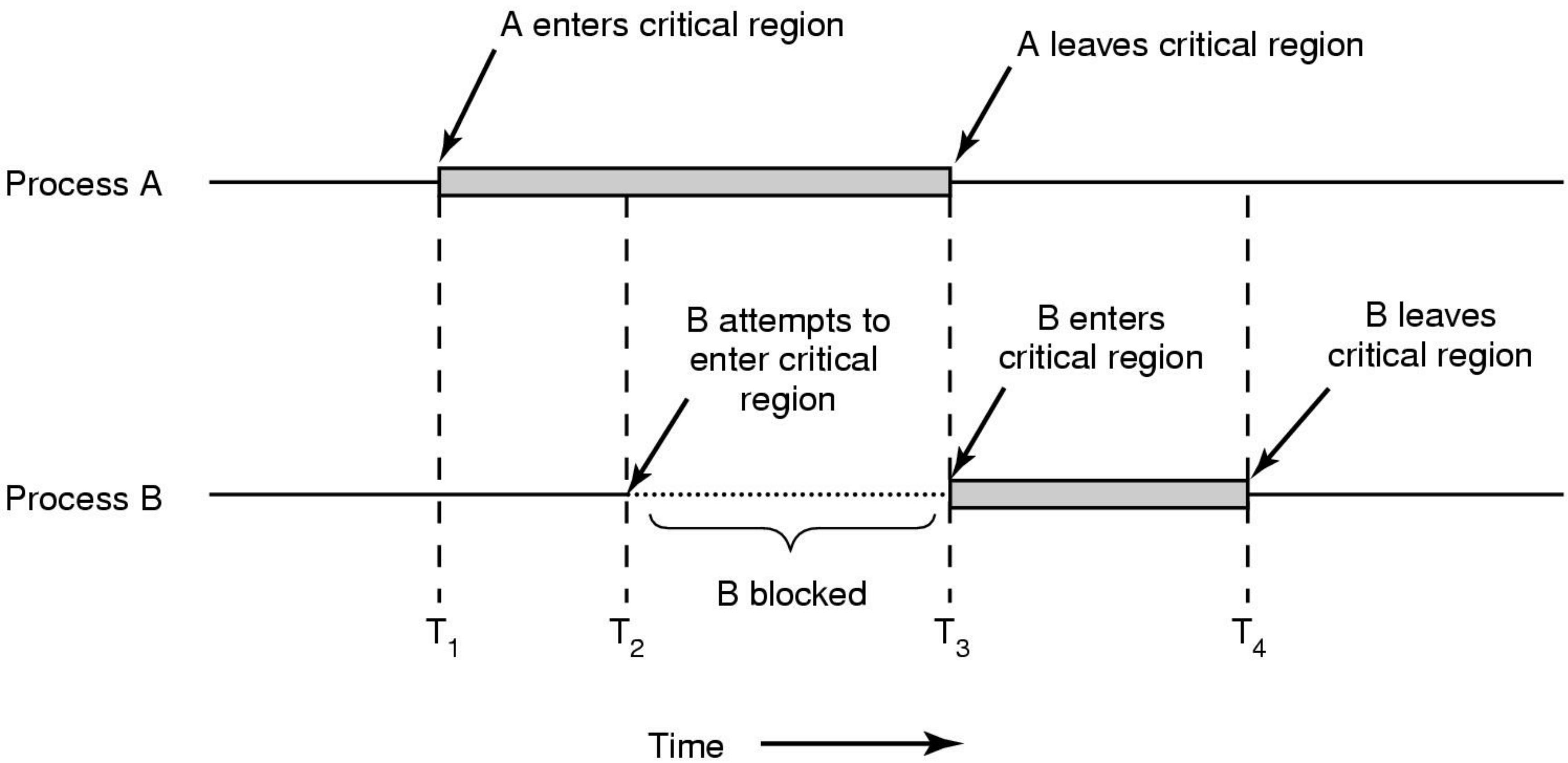
Critical Region

- That part of the program that do critical things such as a**ccessing shared memory**
- Can lead to race condition

Solution Requirement

1) No two processes simultaneously in critical region
2) No assumptions made about speeds or numbers of CPUs
3) No process running outside its critical region may block another process
4) No process must wait forever to enter its critical region

Solution Requirement



Mutual exclusion With Busy Waiting

- Possible Solutions

1. Disabling Interrupts
2. Lock Variables
3. Strict Alternation
4. Peterson’s solution
5. TSL

Disabling Interrupts

- How does it work?
  - Disable all interrupts just after entering a critical section
  - Re-enable them just before leaving it.

```
while (true) {
    /* disable interrupts *
    /* critical section */;
    /* enable interrupts */
    /* remainder */;
}
```

- Why does it work?
  - With interrupts disabled, no clock interrupts can occur
  - No context switching can occur
- Problems:
  - What if the process forgets to enable the interrupts?
  - Multiprocessor? (disabling interrupts only affects one CPU)
- Only used inside OS

Lock Variables

```
int lock = 0;
while (lock);
lock = 1;
 //EnterCriticalSection;
   access shared variable;
//LeaveCriticalSection;
lock = 0;
```

Does the above code work?

Ans: No. Race condition still possible. Two processes might read lock == 0 at the same time before either sets it.

What Actually Happens:

Process 1 checks lock == 0 — it’s true.

Before Process 1 sets lock = 1, the CPU **switches to Process 2**.

Process 2 also checks lock == 0 — still true.

Process 2 sets lock = 1, and enters critical section.

Then Process 1 resumes and **also sets lock = 1**, entering the critical section **at the same time**.

Lock Variables

A shared variable (lock) indicates if the critical section is **occupied**.

```
int lock = 0;
while (lock);
Check again here?
lock = 1;
//EnterCriticalSection;
  access shared variable;
//LeaveCriticalSection;
lock = 0;
```

Still doesn’t work!

Strict Alternation

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

a) Process 0 **(b) Process 1**

Proposed solution to critical region problem

Problem

- Busy waiting: Continuously testing a variable until some value appear
  - Wastes CPU time
- Violates condition 3
  - When one process is much slower than the other

Peterson's solution

- Consists of 2 procedures
- Each process has to call
  - `enter_region` with its own process # before entering its C.R.
  - And `Leave_region` after leaving C.R.

```
do{
    enter_region(process#)
        critical section
    leave_region(process#)
        remainder section
} while (TRUE);
```

Peterson's solution (for 2 processes)

```
#define FALSE  0
#define TRUE   1
#define N      2                          /* number of processes */

int turn;                                 /* whose turn is it? */
int interested[N];                        /* all values initially 0 (FALSE) */

void enter_region(int process);           /* process is 0 or 1 */
{
     int other;                           /* number of the other process */

     other = 1 – process;                 /* the opposite of process */
     interested[process] = TRUE;          /* show that you are interested */
     turn = process;                      /* set flag */
     while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)            /* process: who is leaving */
{
     interested[process] = FALSE;         /* indicate departure from critical region */
}
```

Peterson's Solution: Analysis(1)

- Let Process 1 is not interested and Process 0 calls enter_region with 0
- So, turn = 0 and interested[0] = true and Process 0 is in CR
- Now if Process 1 calls enter_region, it will hang there until interested[0] is false. Which only happens when Process 0 calls leave_region i.e. leaves the C.R.

Peterson's Solution: Analysis(2)

- Let both processes call enter_region simultaneously
- Say turn = 1. (i.e. Process 1 stores last)
- Process 0 enters critical region: while (turn = = 0 && …) returns false since turn = 1.
- Process 1 loops until process 0 exits: while (turn = = 1 && interested[0] = = true) returns true.
- It works fine!!

Busy Waiting: Problems

- Waste CPU time since it sits on a tight loop
- May have unexpected effects:
  - **Priority Inversion Problem**

Example:

- 2 Cooperating Processes: H (high priority) and L (low priority)
- *Scheduling rule:* H is run whenever it is ready
- Let L in C. R. and H is ready and wants to enter C.R.
- Since H is ready it is given the CPU and it starts busy waiting
- L will never gets the chance to leave its C.R.
- H loops forever

Sleep & wakeup

- When a process has to wait, change its state to **BLOCKED**
- Switched to **READY** state, when it is OK to retry entering the critical section
- Sleep is a system call that causes the caller to block
  - be suspended until another process wakes it up
- Wakeup system call has one parameter, the process to be awakened.
- Let's illustrate the use of sleep & wakeup with an example: The producer consumer problem

Producer Consumer Problem

- Also called bounded-buffer problem
- Two (or *m*+*n*) processes share a common buffer
- One (or *m*) of them is (are) **producer**(s): put(s) information in the buffer
- One (or *n*) of them is (are) **consumer**(s): take(s) information out of the buffer
- Trouble and solution
  - Producer wants to put but buffer full- Go to **sleep** and **wake up** when consumer takes one or more
  - Consumer wants to take but buffer empty- go to sleep and wake up when producer puts one or more

Sleep and Wakeup

```
#define N 100                                      /* number of slots in the buffer */
int count = 0;                                     /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                                /* repeat forever */
          item = produce_item( );                  /* generate next item */
          if (count == N) sleep( );                /* if buffer is full, go to sleep */
          insert_item(item);                       /* put item in buffer */
          count = count + 1;                       /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);        /* was buffer empty? */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                                /* repeat forever */
          if (count == 0) sleep( );                /* if buffer is empty, got to sleep */
          item = remove_item( );                   /* take item out of buffer */
          count = count − 1;                       /* decrement count of items in buffer */
          if (count == N − 1) wakeup(producer);  /* was buffer full? */
          consume_item(item);                      /* print item */
     }
}
```

Sleep and Wakeup

```
#define N 100                               /* number of slots in the buffer */
int count = 0;                              /* number of items in the buffer */
```

```
void producer(void)
{
     int item;

     while (TRUE) {
          item = produce_item( );
          if (count == N) sleep( );
          insert_item(item);
          count = count + 1;
          if (count == 1) wakeup(consumer);
     }
}
```

```
void consumer(void)
{
     int item;

     while (TRUE) {                                    /
          if (count == 0) sleep( );                    /
          item = remove_item( );                       /
          count = count − 1;                           /
          if (count == N − 1) wakeup(producer);
          consume_item(item);                          /
     }
}
```

Producer-consumer problem

Sleep and Wakeup: Race condition

- **Busy waiting problem is resolved but the following race condition exists**
- Unconstrained access to *count*
  - CPU is given to P just after C has read count to be 0 but not yet gone to sleep.
  - P calls wakeup
  - Result is lost wake-up signal
  - Both will sleep forever

Semaphores

- A new variable type
- A kind of generalized lock
  - First defined by Dijkstra in late 60s
  - Main synchronization primitive used in original UNIX
- Semaphores are like integers, except
  - No negative values
  - Only operations allowed are *up* and *down* – can't read or write value, except to set it initially

Semaphores: Types

- **Counting semaphore**.
  - The value can range over an unrestricted domain
- **Binary semaphore**
  - The value can range only between 0 and 1.
  - On some systems, binary semaphores are known as mutex locks as they provide mutual exclusion

Semaphores: Operation

- Operation "down":
  - if value > 0; value-- and then continue.
  - if value = 0; process is put to sleep without completing the down for the moment
    - Checking the value, changing it, and possibly going to sleep, is all done as an ***atomic*** *action*.
- Operation "up":
  - increments the value of the semaphore addressed.
  - If one or more process were sleeping on that semaphore, one of them is chosen by the system (e.g. at random) and is allowed to complete its *down*
    - The operation of incrementing the semaphore and waking up one process is also indivisible
  - No process ever blocks doing an *up.*

Semaphores: Atomicity

- Operations must be atomic
  - Two *down*'s together can't decrement value below zero
  - Similarly, process going to sleep in *down* won't miss wakeup from *up* – even if they both happen at same time

Producer & consumer

```c
#define N 100                      /* number of slots in the buffer */
typedef int semaphore;             /* semaphores are a special kind of int */
semaphore mutex = 1;               /* controls access to critical region */
semaphore empty = N;               /* counts empty buffer slots */
semaphore full = 0;                /* counts full buffer slots */
```

```c
void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item( );
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}
```

```c
void consumer(void)
{
    int item;

    while (TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item( );
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```

Semaphores in Producer Consumer Problem: Analysis

- 3 semaphores are used
  - *full* (initially 0) for counting occupied slots
  - *Empty* (initially *N*) for counting empty slots
  - *mutex* (initially 1) to make sure that Producer and Consumer do not access the buffer at the same time.
- Here 2 uses of semaphores
  - Mutual exclusion (mutex)
  - Synchronization (full and empty)
    - To guarantee that certain event sequences do or do not occur

Block on:	Unblock on:
Producer: insert in full buffer	Consumer: item inserted
Consumer: remove from empty buffer	Producer: item removed

Semaphores: Usage

1. Mutual exclusion-mutex
2. Controlling access to limited resource
3. Synchronization

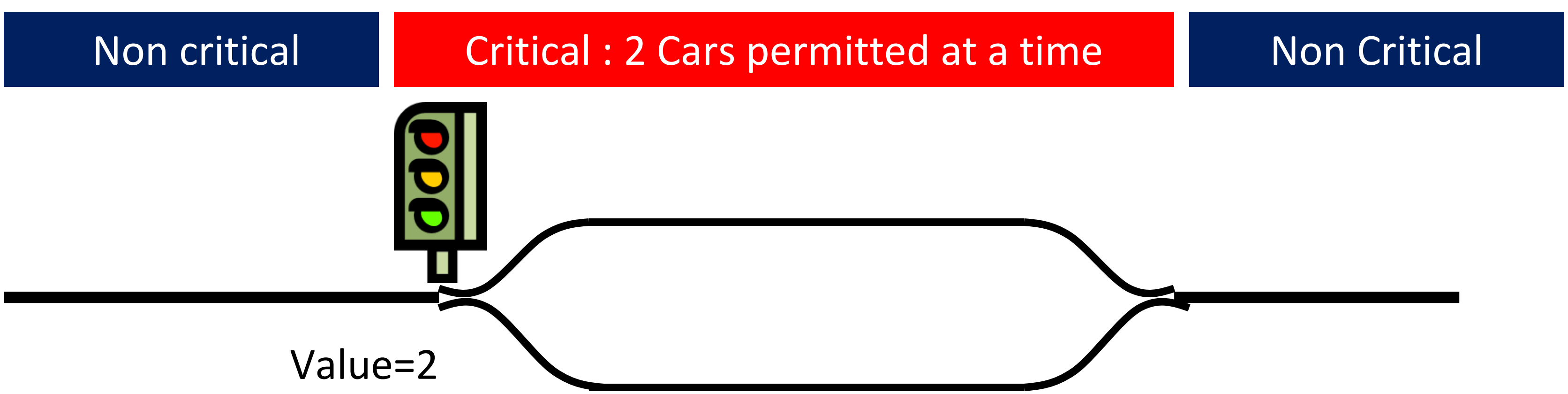
Mutual exclusion

- How to ensure that only one process can enter its C.R.?
- Binary semaphore initialized to 1
- Shared by all collaborating processes
- If each process does a *down* just before entering CR and an *up* just after leaving , then mutual exclusion is guaranteed

```
do {
        down(mutex);
        // critical section
        up(mutex);
        // remainder section
} while (TRUE);
```

Controlling access to a resource

- What if we want maximum **m** process/thread can use a resource simultaneously ?
- Counting semaphore initialized to the number of available resources
- Semaphore from railway analogy
  - Here is a semaphore **initialized to 2** for resource control:



Synchronization

- How to resolve dependency among processes
- Binary semaphore initialized to 0
- consider 2 concurrently running processes:
  - P1 with a statement S1 and
  - P2 with a statement S2.
  - Suppose we require that S2 be executed only after S1 has completed.

P1

```
S1;
up(synch);
```

P2

```
down(synch);
S2;
```

Semaphores: “Be Careful”

- Suppose the following is done in Producer’s code

```
…
down(&empty)
down(&mutex)
…
```

```
…
down(&mutex)
down(&empty)
…
```

Just the order is reversed

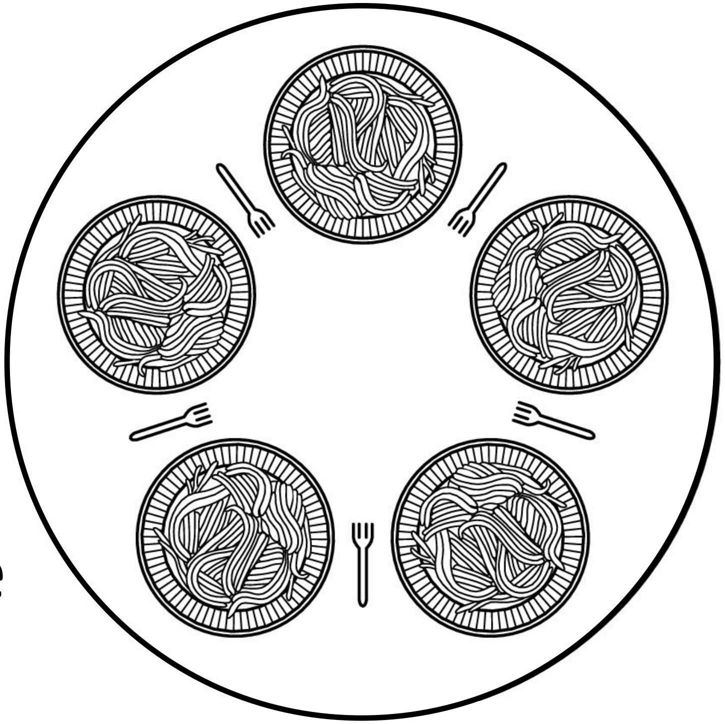
- If buffer full P would block due to down(&empty) with mutex = 0.
- So now if C tries to access the buffer, it would block too due to its down(&mutex).
- Both processes would stay blocked forever: DEADLOCK

Barriers

- A synchronizing mechanism intended for groups of processes
- Some applications are divided into phases and have the rule that no process may proceed into the next phase until all processes are ready to proceed to the next phase.
- This can be done by placing a barrier at the end of each phase.
- When a process reaches the barrier, it is blocked until all processes have reached the barrier.

Dining Philosophers

- Philosophers spend their lives thinking and eating
- Don’t interact with their neighbors
- When get hungry try to pick up 2 chopsticks (one at a time in either order) to eat
- Need both to eat, then release both when done
- How to program the scenario avoiding all concurrency problems?



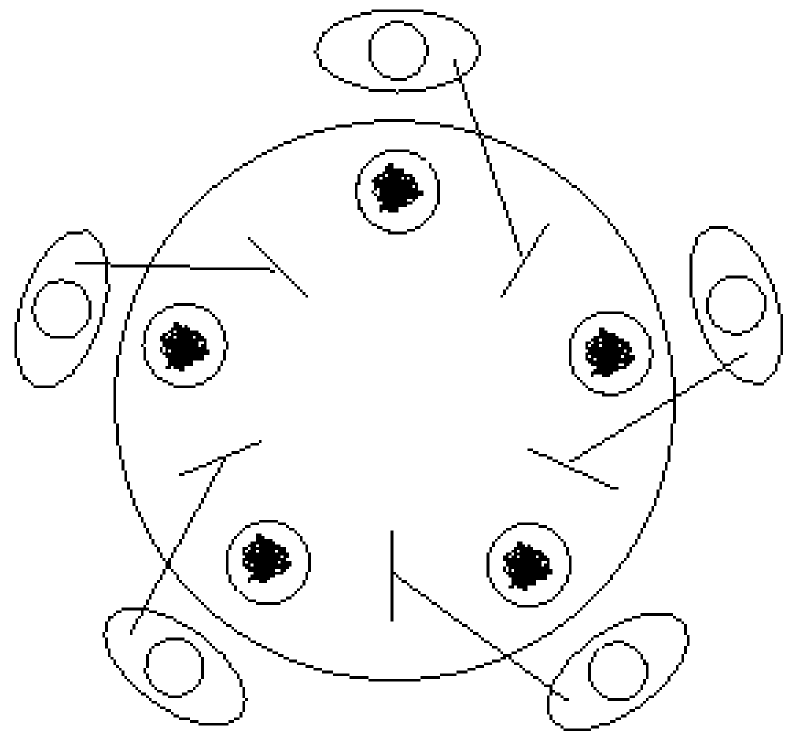
Dining Philosophers: A Solution

```
#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 */
    }
}
```

Dining Philosophers: Problems with Previous Solution

- Deadlock may happen
- Does this solution prevents any such thing from happening ?
  - Everyone takes the left fork simultaneously



Dining Philosophers: Problems with Previous Solution

Tentative Solution:

- After taking left fork, check whether right fork is available.
- If not, then return left one, wait for some time and repeat again.

Problem:

- All of them start and do the algorithm synchronously and simultaneously:
- STARVATION (A situation in which all the programs run indefinitely but fail to make any progress)
- Solution: Random wait; but what if the most unlikely of same random number happens?

Another Attempt, Successful!

```
void philosopher(int i)
{
  while (true)
  {
      think();
      down(&mutex);
      take_fork(i);
      take_fork((i+1)%N);
      eat();
      put_fork(i);
      put_fork((i+1)%N);
      up(&mutex);
  }
}
```

- Theoretically solution is OK- no deadlock, no starvation.
- Practically with a performance bug:
  - Only one philosopher can be eating at any instant: absence of parallelism

How It Solves the Problem:

1. Prevents Deadlock

In the basic algorithm (without mutex), all philosophers could:

- Pick up their left fork.
- Wait forever for the right fork (which their neighbor has).

Result: deadlock.

Using mutex:

Only one philosopher can execute the section between down() and up() at a time.

- So, only one can pick up both forks and eat.
- Others are blocked at down(&mutex) and wait their turn.

→ No simultaneous fork grabbing = no circular wait = no deadlock.

Final Solution part 1

```
#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 */
     }
}
```

Final Solution Part 2

```
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]);
    }
}
```

Thanks for your sincerity