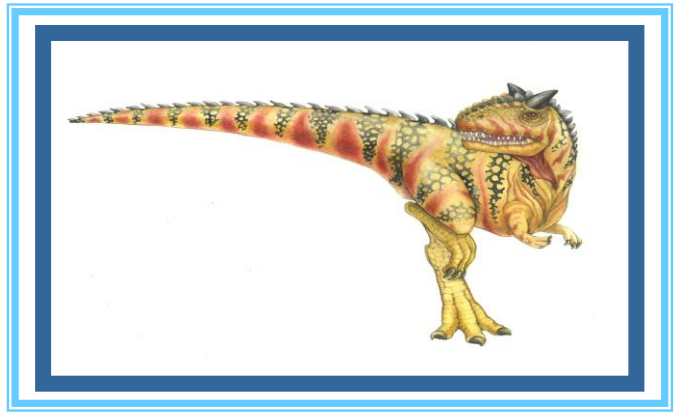
Chapter 5: Process Synchronization



Chapter 5: Process Synchronization

- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Mutex Locks
- Semaphores
- Classic Problems of Synchronization
- Monitors
- Synchronization Examples
- Alternative Approaches

Objectives

- To present the concept of process synchronization.
- To introduce the **critical-section problem,** whose solutions can be used to ensure the consistency of shared data
- To present both **software and hardware solutions** of the critical-section problem
- To examine several classical process-synchronization problems
- To explore several tools that are used to solve process synchronization problems

Recap & Looking ahead

Processes

Independent

Cooperative

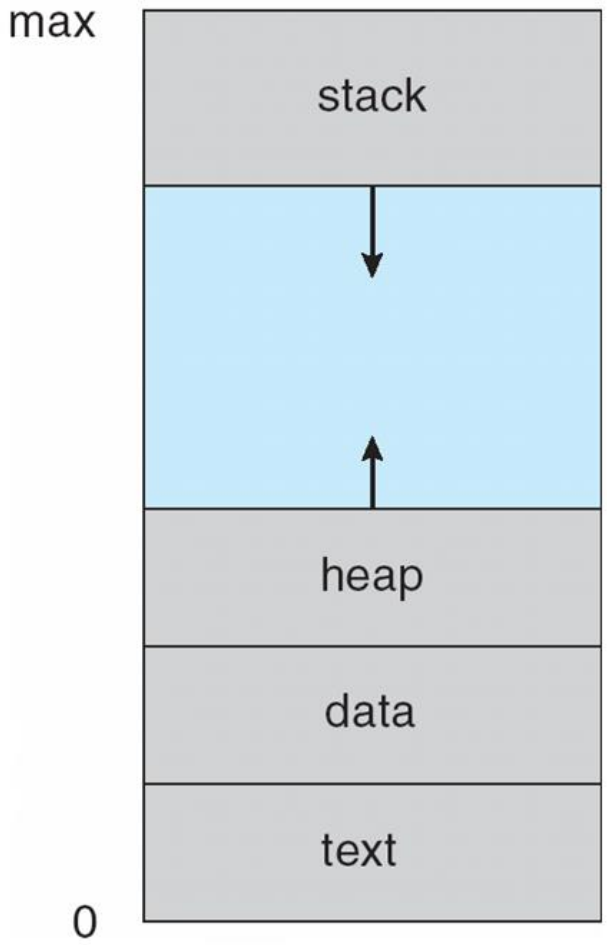
IPC
↳ Shared mem + Message passing

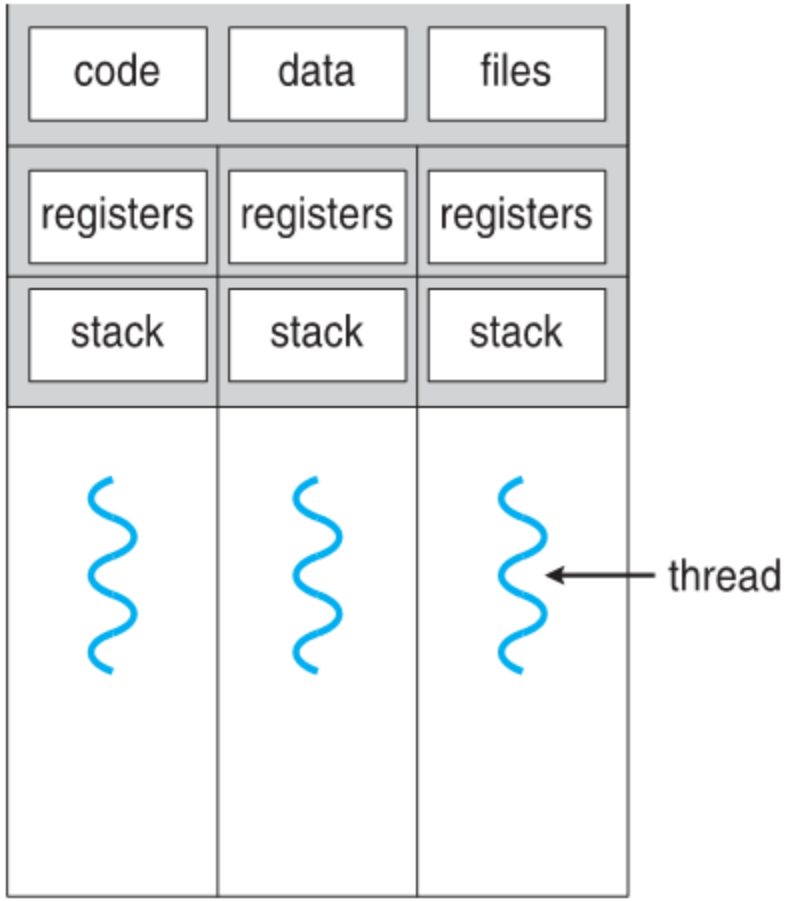
That affect or are affected by other processes executing in the system

Share
code + data → files | messages

↳ Concurrent access → data inconsistency

Recap & Looking ahead





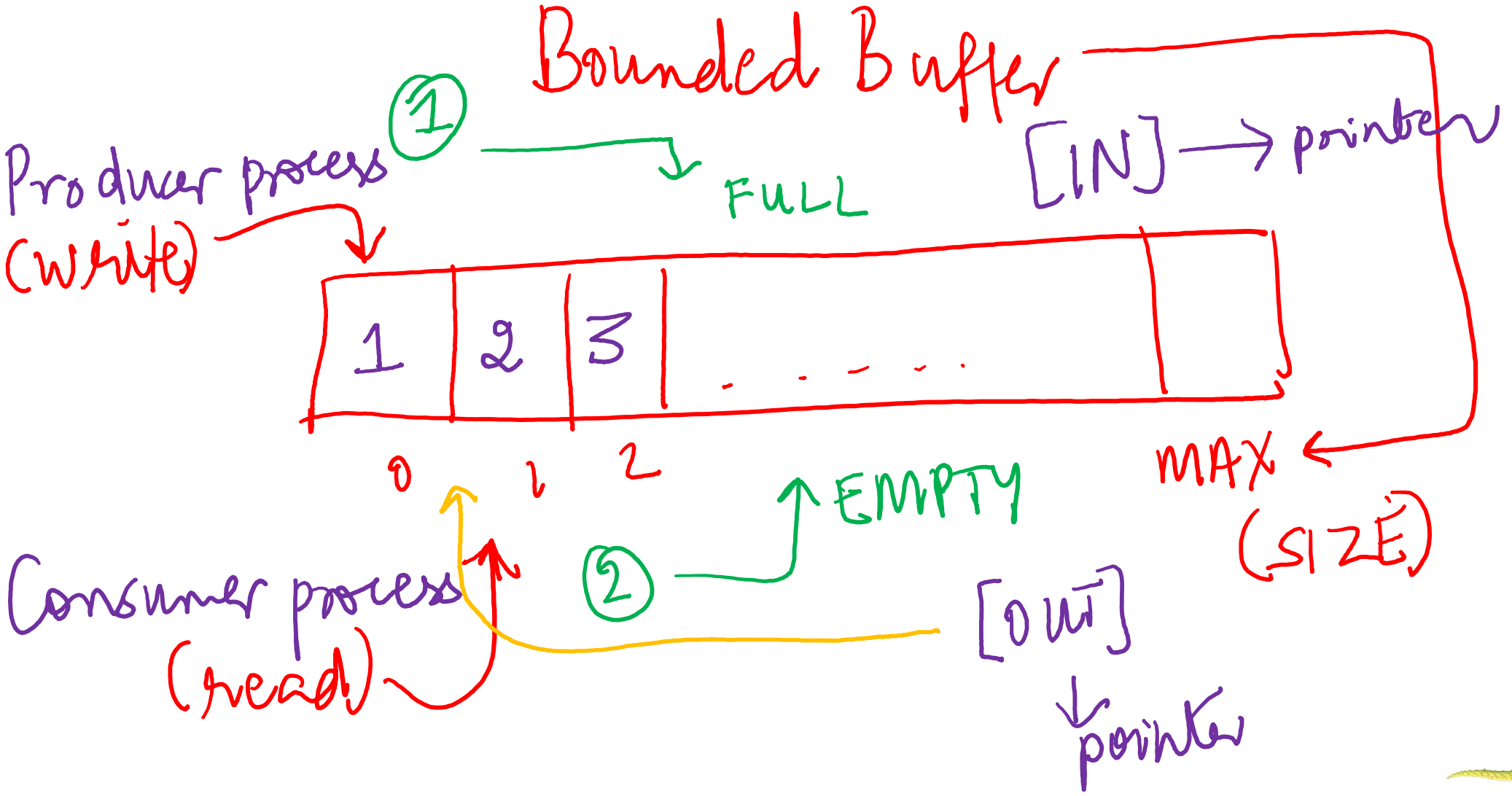
multithreaded process

Background

- Processes can **execute concurrently**
  - May ***be interrupted at any time***, partially completing execution
- **Concurrent access to shared data may result in data inconsistency**
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Illustration of the problem:
- Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffers. We can do so by having an integer counter that keeps track of the number of full buffers. Initially, counter is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Recap on Producer Consumer Problem

COUNTER



Producer

```
while (true) {
        /* produce an item in next produced */

        while (counter == BUFFER_SIZE) ;
                /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

Consumer

```
while (true) {
        while (counter == 0)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        counter--;
        /* consume the item in next consumed */
}
```

Race Condition

- **`counter++`** could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- **`counter--`** could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Consider this execution interleaving with "count = 5" initially:

S0: producer execute `register1 = counter`	{register1 = 5}
S1: producer execute `register1 = register1 + 1`	{register1 = 6}
S2: consumer execute `register2 = counter`	{register2 = 5}
S3: consumer execute `register2 = register2 - 1`	{register2 = 4}
S4: producer execute `counter = register1`	{counter = 6 }
S5: consumer execute `counter = register2`	{counter = 4}

Critical Section Problem

- Consider system of ***n*** processes {p_0, p_1, … p_{n-1}}
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- ***Critical section problem*** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

Critical Section

- General structure of process P_i

```
do {
        entry section
            critical section
        exit section
            remainder section
} while (true);
```

Solution to Critical-Section Problem

1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
   - Assume that each process executes at a nonzero speed
   - No assumption concerning **relative speed** of the ***n*** processes

Algorithm for Process P_i

```
do {

    while (turn == j);

          critical section

    turn = j;

          remainder section

} while (true);
```

Algorithm for Process P_i

```
do {

    while (turn == j);

        critical section

    turn = j;

        remainder section

 } while (true);
```

turn = j
P_i → enter CS

turn is other process

turn != i

// before return, set turn = j

while (turn == i);
CS
turn = i;
RS

P_i entry is regulated by P_j & vice versa

RECAP QUESTION 1

1. **What do you understand by Race condition?**

A situation where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a race condition.

2. **Busy wait means?**

RECAP QUESTION 2 & 3

3. **What is the solution to race condition?**

   Process synchronization

4. **Critical section problem must satisfy which three conditions?**

   a. Mutual exclusion

   b. Progress

   c. Bounded waiting

Critical section
Entry
CS
Exit
Gen structure

flag variable → 2 process → Peterson

boolean

Progress X

flag[i] = T/F

Pi Pj

Pj → CS

F/T

Entry section

① flag[i] = T;

② while (flag[j] == T);

Critical section

Exit section

flag[i] = F

remainder section

Pj

① flag[j] = T;

② while (flag[i] == T);

Critical section

flag[j] = F

remainder section

flag ←

Pi Pj who enters CS?

T	T	Pi
T	F	Pi
F	T	Pj
F	F	X

Flag[i] = ~~F~~ T

Flag[j] = ~~F~~ T

Pi: ① | Pj: ① ② | Pi while

{while}

Deadlock!

Critical-Section Handling in OS

HOW?

→ 2 processes → open files simultaneously?

Two approaches depending on if kernel is preemptive or non-preemptive

- **Preemptive** – allows preemption of process when running in kernel mode
- **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU
  - Essentially free of race conditions in kernel mode

Execution in Kernel mode → OS designer!

Peterson's Solution

- Good algorithmic description of solving the problem
- Two process solution
- Assume that the **`load`** and **`store`** machine-language instructions are **atomic**; that is, ***cannot be interrupted***
- The two processes share two variables:
  - **`int turn;`**
  - **`Boolean flag[2]`**
- The variable **`turn`** indicates whose turn it is to enter the critical section
- The **`flag`** array is used to indicate if a process is ready to enter the critical section. **`flag[i] = true`** implies that process $\mathbf{P_i}$ is ready!

Algorithm for Process P_i

Initially flag[i] = F
flag[j] = F

P_i wishes to enter CS & raises its flag!

```
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = false;

        remainder section

} while (true);
```

→ and sets turn = j // next process

P_i → false && → False

P_i enters critical section

→ checks next process!

re-sets flag = F on exit!

Algorithm for Process P_J

```
do {
    flag[j] = true;
    turn = i;
    while (flag[i] && turn = = i);

        critical section

    flag[j] = false;

        remainder section

} while (true);
```

Algorithm for Process P_i

```
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn = = j);
        critical section
    flag[i] = false;
        remainder section
} while (true);
```

Pi → wants to enter

Suppose Pj → wishes, turn = i

Pi enters

Continuity → Mutual exclusion

Progress ✓

At most one time, wait!

Peterson's Solution (Cont.)

- Provable that the three CS requirement are met:
  1. Mutual exclusion is preserved
     P_i enters CS only if:
     either **`flag[j] = false`** or **`turn = i`**
  2. Progress requirement is satisfied
  3. Bounded-waiting requirement is met

Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- All solutions below based on idea of **locking**
  - Protecting critical regions via locks
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
    - **Atomic** = non-interruptible
  - Either test memory word and set value
  - Or swap contents of two memory words

Solution to Critical-section Problem Using Locks

```
do {
    acquire lock
        critical section
    release lock
        remainder section
} while (TRUE);
```

Atomic

acquire lock → entry section

release lock → exit section

test_and_set Instruction

Definition:

```
boolean test_and_set (boolean *target)
    {
        boolean rv = *target;
        *target = TRUE;
        return rv;
    }
```

1. Executed atomically
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to “TRUE”.

Solution using test_and_set()

- Shared Boolean variable lock, initialized to FALSE
- Solution:

```
do {
    while (test_and_set(&lock)) /*while true*/
        ; /* do nothing */

        /* critical section */

    lock = false;

        /* remainder section */

} while (true);
```

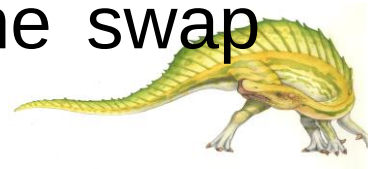
compare_and_swap Instruction

Definition:

```
int compare _and_swap(int *value, int expected, int new_value) {
    int temp = *value;

    if (*value == expected)
       *value = new_value;
 return temp;
}
```

1. Executed atomically
2. Returns the original value of passed parameter “value”
3. Set the variable “value” the value of the passed parameter “new_value” but only if “value” ==“expected”. That is, the swap takes place only under this condition.



Solution using compare_and_swap

- Shared integer “lock” initialized to 0;
- Solution:

```
do {
    while (compare_and_swap(&lock, 0, 1) != 0)
        ; /* do nothing */

    /* critical section */

    lock = 0;

    /* remainder section */

} while (true);
```

GLOBAL VARIABLE INITIALISED TO ZERO

BREAK.

How to satisfy all critical section requirements?

- Hardware
- Software

Bounded-waiting Mutual Exclusion with test_and_set

```
do {
    waiting[i] = true;
    key = true;
    while (waiting[i] && key)
        key = test_and_set(&lock);
    waiting[i] = false;

    /* critical section */

    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;

    if (j == i)
        lock false;
    else
        waiting[j] = false;

    /* remainder section */
} while (true);
```

Mutex Locks (MUTUAL EXCLUSION)

- Previous **hardware solutions** are complicated and generally inaccessible to ***application programmers***
- **OS designers** build software tools to solve critical section problem
- **Simplest is mutex lock**
  - Protect a critical section by first ***acquire()*** a lock then ***release()*** the lock
  - Prevents what? RACE CONDITION
  - Boolean variable ***(available)*** indicating if lock is available or not

Mutex Locks

- Calls to **acquire()** and **release()** must be atomic
  - Usually implemented via hardware atomic instructions
- But this solution **requires busy waiting**
  - This lock therefore is called a **spinlock**

acquire() and release()

```
acquire() {
   while (!available)
      ; /* busy wait */
   available = false;
}

release() {
   available = true;
}
```

```
do {
    acquire lock
        critical section
    release lock
        remainder section
} while (true);
```

Semaphore (S/W solution)

- Synchronization tool that provides more sophisticated ways (not Mutex locks) for process to synchronize ativities -- > Semaphore ***S*** – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - **`wait()`** and **`signal()`**
- Definition of the **`wait()` operation**

```
wait(S) {
        while (S <= 0)
            ; // busy wait
        S--;
}
```

- Definition of the **`signal()` operation**

```
signal(S) {
                    S++;
}
```

Entry → wait

↓ Semaphore

Exit → signal

↑ Semaphore

Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock**
- **Can solve** various synchronization problems
- Consider P_1 and P_2 that require S_1 to happen before S_2
  Create a semaphore "**syn**" initialized to 0

```
P1:
    S1;
    signal(syn);
P2:
    wait(syn);
    S2;
```

- Can implement a counting semaphore ***S*** as a binary semaphore

Semaphore Implementation

- Must guarantee that no two processes can execute the **`wait()`** and **`signal()`** on the same semaphore at the same time
- Thus, the implementation becomes the critical section problem where the **`wait`** and **`signal`** code are placed in the critical section
  - Could now have **busy waiting** in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list
- Two operations:
  - **block** – place the process invoking the operation on the appropriate waiting queue
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue
-
```
typedef struct{
  int value;
  struct process *list;
  } semaphore;
```

(Alternative)

To overcome Busy wait

Implementation with no Busy waiting (Cont.)

```
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        block();
    }
}
```

```
signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
    }
}
```

Deadlock and Starvation

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let ***S*** and ***Q*** be two semaphores initialized to 1

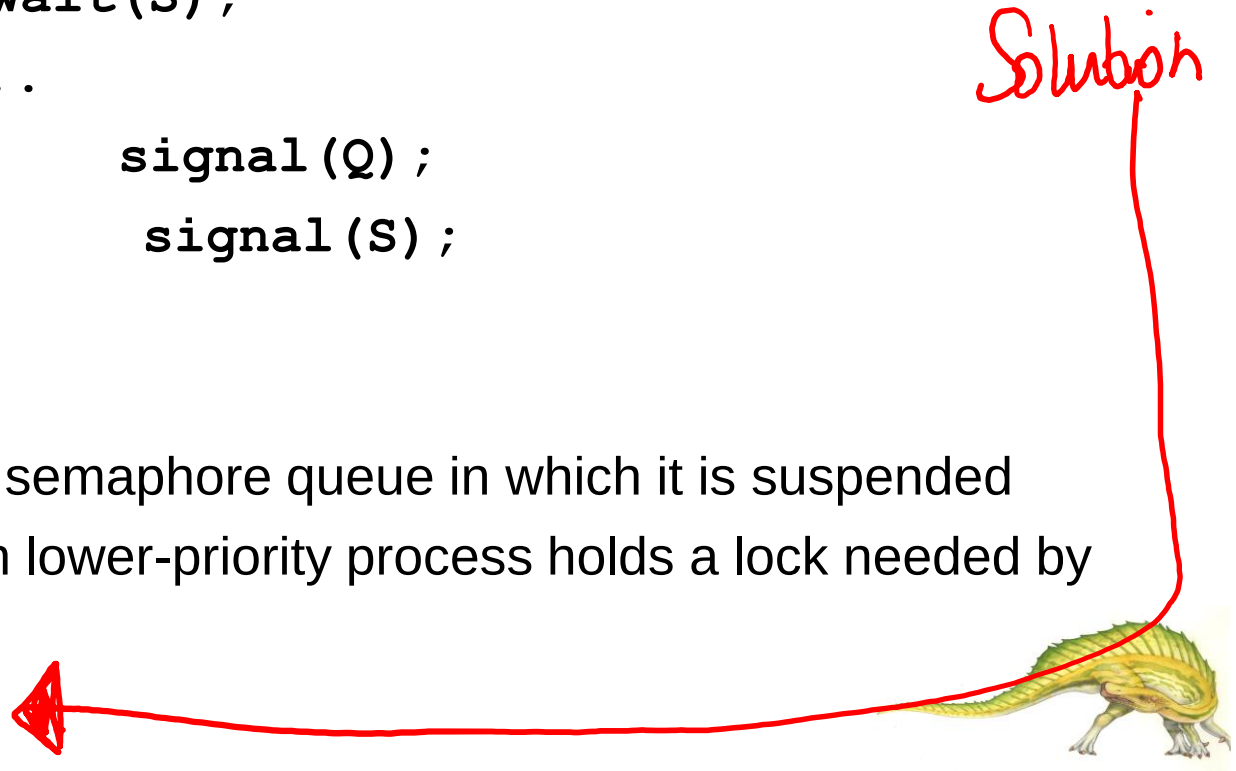
P_0

```
wait(S);
 wait(Q);
...
  signal(S);
   signal(Q);
```

P_1

```
wait(Q);
 wait(S);
...
      signal(Q);
       signal(S);
```

- **Starvation** – **indefinite blocking**
  - A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via **priority-inheritance protocol**



Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

Bounded-Buffer Problem

- n buffers, each can hold one item
- Semaphore mutex initialized to the value 1
- Semaphore full initialized to the value 0
- Semaphore empty initialized to the value n

Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
do {
        ...
        /* produce an item in next_produced */
        ...
    wait(empty);
    wait(mutex);
        ...
        /* add next produced to the buffer */
        ...
    signal(mutex);
    signal(full);
} while (true);
```

Bounded Buffer Problem (Cont.)

- Structure of the consumer process

```
do {
    wait(full);
    wait(mutex);
        ...
/* remove an item from buffer to next_consumed */
        ...
    signal(mutex);
    signal(empty);
        ...
    /* consume the item in next consumed */
        ...
} while (true);
```

BREAK!

Readers-Writers Problem

- A **data set** is **shared** among a **number of concurrent processes**
  - **Readers:** only read data set; they do ***not*** perform any updates
  - **Writers:** can both read and write
- Problem: allow multiple readers to read at the same time
  - Only one single writer can access shared data at the same time
- Several variations of how readers and writers are considered – all involve some form of priorities
- Shared Data
  - Data set
  - Semaphore **`rw_mutex`** initialized to 1
  - Semaphore **`mutex`** initialized to 1
  - Integer **`read_count`** initialized to 0

Readers-Writers Problem - Solution

- The structure of a **writer** process

```
do {
    wait(rw_mutex);
        ...
    /* writing is performed */
        ...
    signal(rw_mutex);
} while (true);
```

0

rw=1; rw<=0 false

Readers-Writers Problem - Solution

Writer 1

```
do {
        wait(rw_mutex);
        ...
      /* writing is performed */
              ...
         signal(rw_mutex);
    } while (true);
```

Writer 2

```
do {
        wait(rw_mutex);
        ...
      /* writing is performed */
              ...
         signal(rw_mutex);
    } while (true);
```

rw<0 ? NO

rw--

rw<0 busy wait

writing over

rw=0

Readers-Writers Problem (Cont.)

- Structure of a **reader** process:

```
do {
    wait(mutex);
    read_count++;
    if (read_count == 1)
        wait(rw_mutex); //first reader
    signal(mutex);
        ...
    /* reading is performed */
        ...
    wait(mutex);
    read_count--;
    if (read_count == 0) // last reader
    signal(rw_mutex);  //last reader
    signal(mutex);
} while (true);
```

Readers-Writers Problem Variations

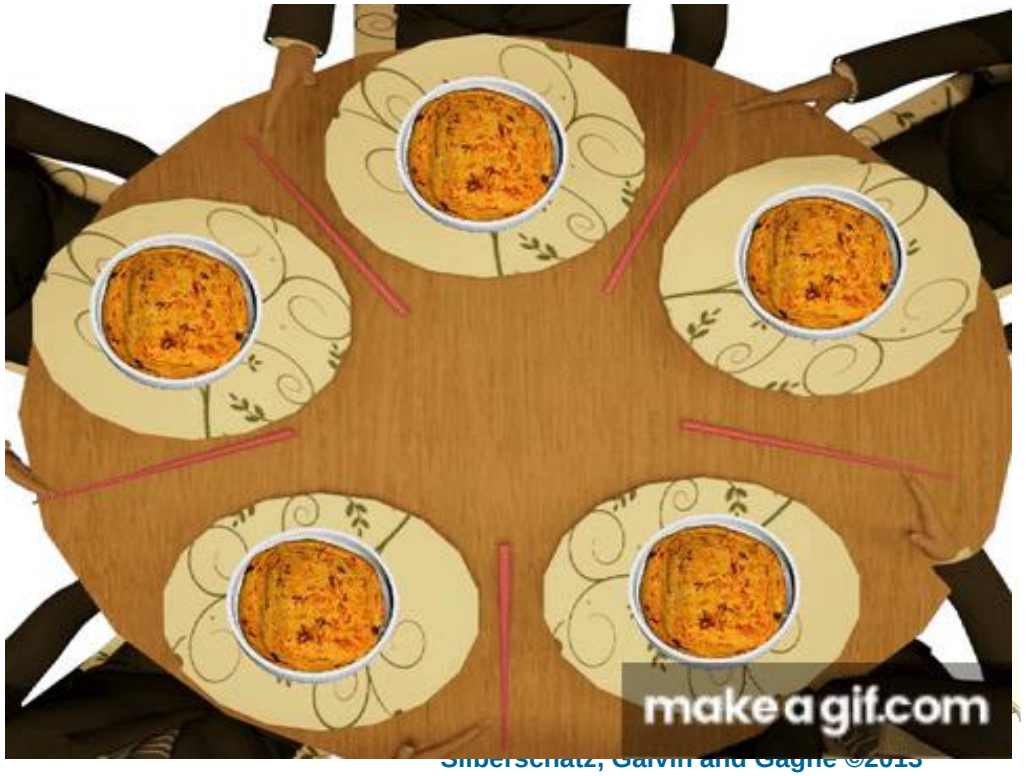
- First variation – no reader kept waiting unless writer has permission to use shared object
- Second variation – once writer is ready, it performs the write ASAP
- Both may have starvation leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks

Check for reader 1 & reader 2 ?

Dining-Philosophers Problem

- Philosophers spend their lives alternating thinking and eating
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done

- In the case of 5 philosophers
  Shared data
    Bowl of rice (data set)
    Semaphore chopstick[5]
    initialized to 1



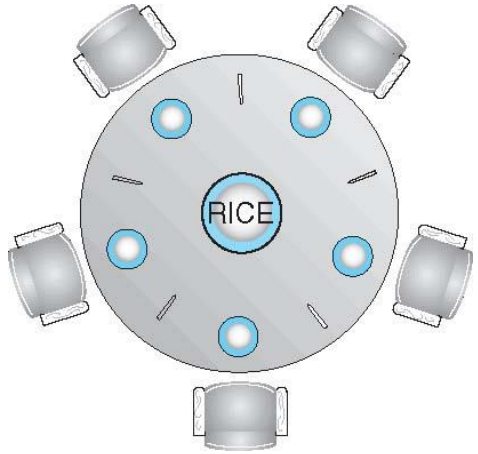
Dining-Philosophers Problem Algorithm

- The structure of **Philosopher *i*:**

```
do {
    wait (chopstick[i] );
    wait (chopStick[ (i + 1) % 5] );
                    //  eat
    signal (chopstick[i] );
    signal (chopstick[ (i + 1) % 5] );

                    //  think

} while (TRUE);
```



What is the problem with this algorithm?

Dining-Philosophers Problem Algorithm (Cont.)

- **Deadlock handling**
  - Allow at most 4 philosophers to be sitting simultaneously at the table.
  - Allow a philosopher to pick up the forks only if both are available (picking must be done in a critical section)
  - Use an asymmetric solution -- an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up first the right chopstick and then the left chopstick.

Problems with Semaphores

- Incorrect use of semaphore operations:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)
- Deadlock and starvation are possible.

End of Chapter 5