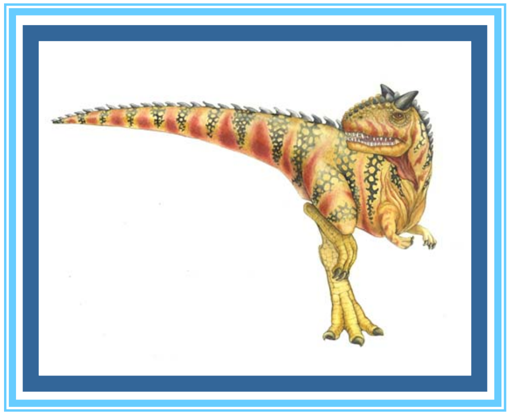
# Chapter 5: Process Synchronization



# 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

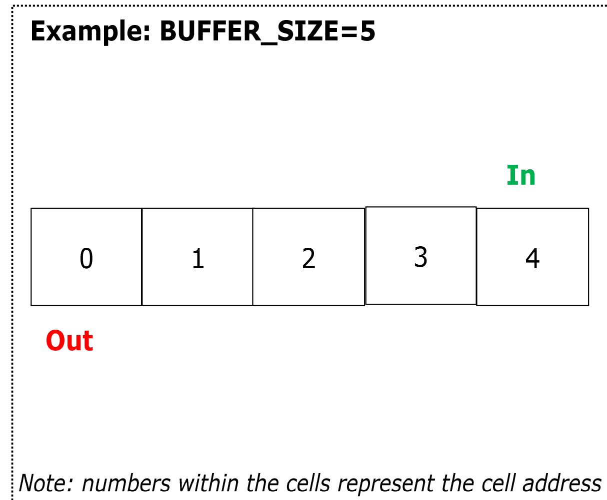
# BACKGROUND

# 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.

# Producer - Consumer

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

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

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



# Race Condition (in Parallelism and Concurrency )

- **`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

# 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**

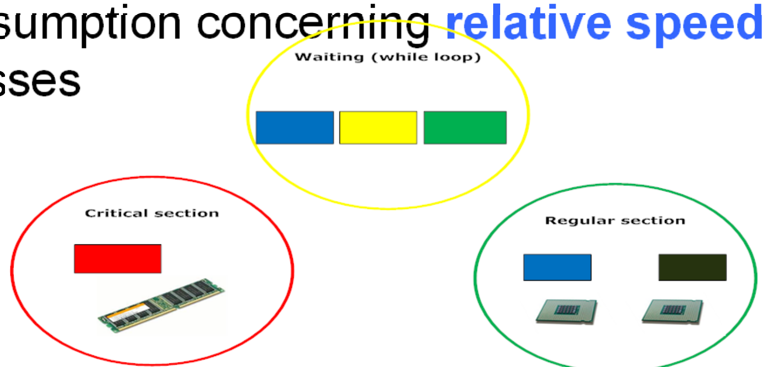
# Critical Section

- General structure of process $\boldsymbol{P_i}$

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

# Solution Criteria

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



# CRITICAL SECTION SOLUTIONS (TECHNIQUES)

# SOFTWARE SOLUTION

# Critical-Section Handling in OS

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

# 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 $P_i$ is ready!

# Peterson's Solution (cont.)

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

        critical section //i.e. counter++

        flag[i] = false;

        remainder section

} while (true);
```

turn: 2

Flag: 1: T, 2: T

P1

```
do {
    flag[1] = true;
    turn = 2;
    while (flag[2] && turn = = 2);

    critical section //i.e. counter++

    flag[1] = false;

    remainder section

} while (true);
```

P2

```
do {
    flag[2] = true;
    turn = 1;
    while (flag[1] && turn = = 1);

    critical section //i.e. counter--

    flag[2] = false;

    remainder section

} while (true);
```

Out of order (memory reordering)
in a modern (Super Scaler) processor that can execute 2 or more instruction at once
I'm talking about single core processor not just multi core processor

1) mov eax, 0
2) mov ebx, 1
3) mov edx, 2
4) inc edx
5) mov ecx, 3
..)

solution 1: (1,2)->(3)->(4,5)

solution 1: (1,2)->(3,5)->(4,..)

The transistors only decide which solution to perform

# 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

- This solution is not visible for modern system architecture because most modern CPUs reorder memory accesses to improve execution efficiency

# HARDWARE SOLUTION

# 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);
```

# 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()

```
BOOLEAN test_and_set(BOOLEAN *oldLock)
{
        BOOLEAN newLock=*oldLock
        *oldLock=true;
        return newLock;
}
```

False

True

Tru

Tru

BOOLEAN lock= False

False

```
do
{
        while(test_and_set(&lock));

        // critical section

        lock=false

        //reminder section

}while(true);
```

False

```
do
{
        while(test_and_set(&lock));

        // critical section

        lock=false

        //reminder section

}while(true);
```

```
BOOLEAN test_and_set(BOOLEAN *oldLock)
{
        BOOLEAN newLock=*oldLock
        *oldLock=true;
        return newLock;
}
```

---

BOOLEAN lock=False

```
do   2
{
        while(test_and_set(&lock));

        // critical section

        lock=false

       //reminder section

}while(true);
```

```
do
{
        while(test_and_set(&lock));

        // critical section

        lock=false

       //reminder section

}while(true);
```

```
do  1
{
        while(test_and_set(&lock));

        // critical section

        lock=false

       //reminder section

}while(true);
```

# 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);
```

# compare_and_swap Instruction

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

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

False

Fals

False

True

True

True

BOOLEAN lock= False

```
do
{
         while (compare_and_swap(&lock, 0, 1) != 0)

         // critical section

         lock=false

        //reminder section

}while(true);
```

```
do
{
         while (compare_and_swap(&lock, 0, 1) != 0)

          // critical section

          lock=false

         //reminder 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.

# Mutex

```
do {
    acquire(lock)   //while(test_and_set(&lock))
        critical section
    release(lock)   //lock=true
        remainder section
} while (true);
```

```
acquire() {
      while (test_and_set(&lock))
         ; /* busy wait */
   }
```

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

# Mutex Locks

- Previous 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
  - Boolean variable indicating if lock is available or not
- Calls to **`acquire()`** and **`release()`** must be atomic
  - Usually implemented via hardware atomic instructions
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**

# SEMAPHORE

# Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore ***S*** – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - **`wait()`** and **`signal()`**
    - Originally called **`P()`** and **`V()`**
- Definition of the **`wait() operation`**

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

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

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

# 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 "**synch**" initialized to 0

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

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

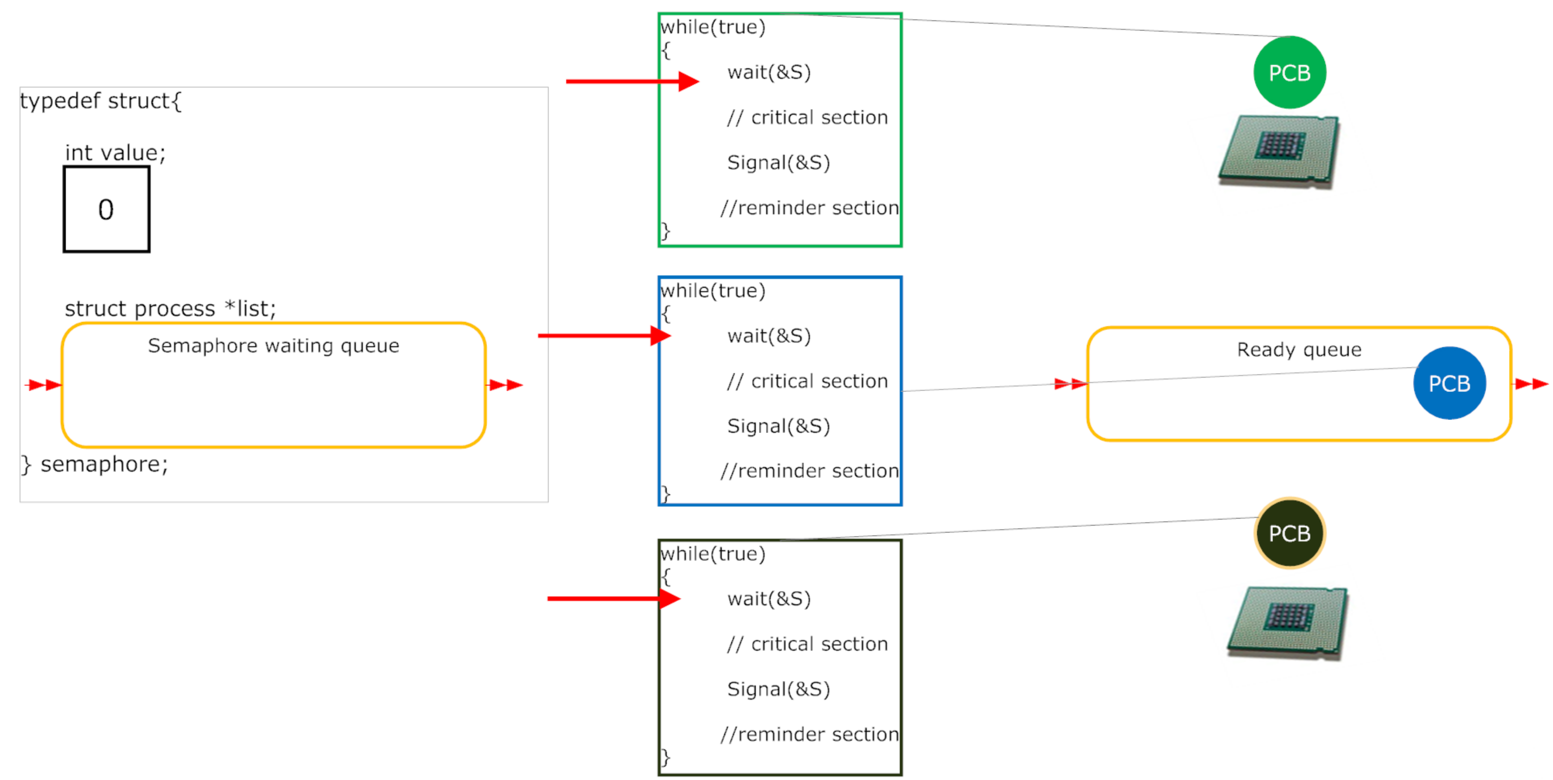
# Semaphore Implementation

- 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

```
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);
     }
}
```



# 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;
```

# PROBLEMS FROM SYNCHRONIZATION TECHNIQUES

# Deadlock

- **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

$P_0$

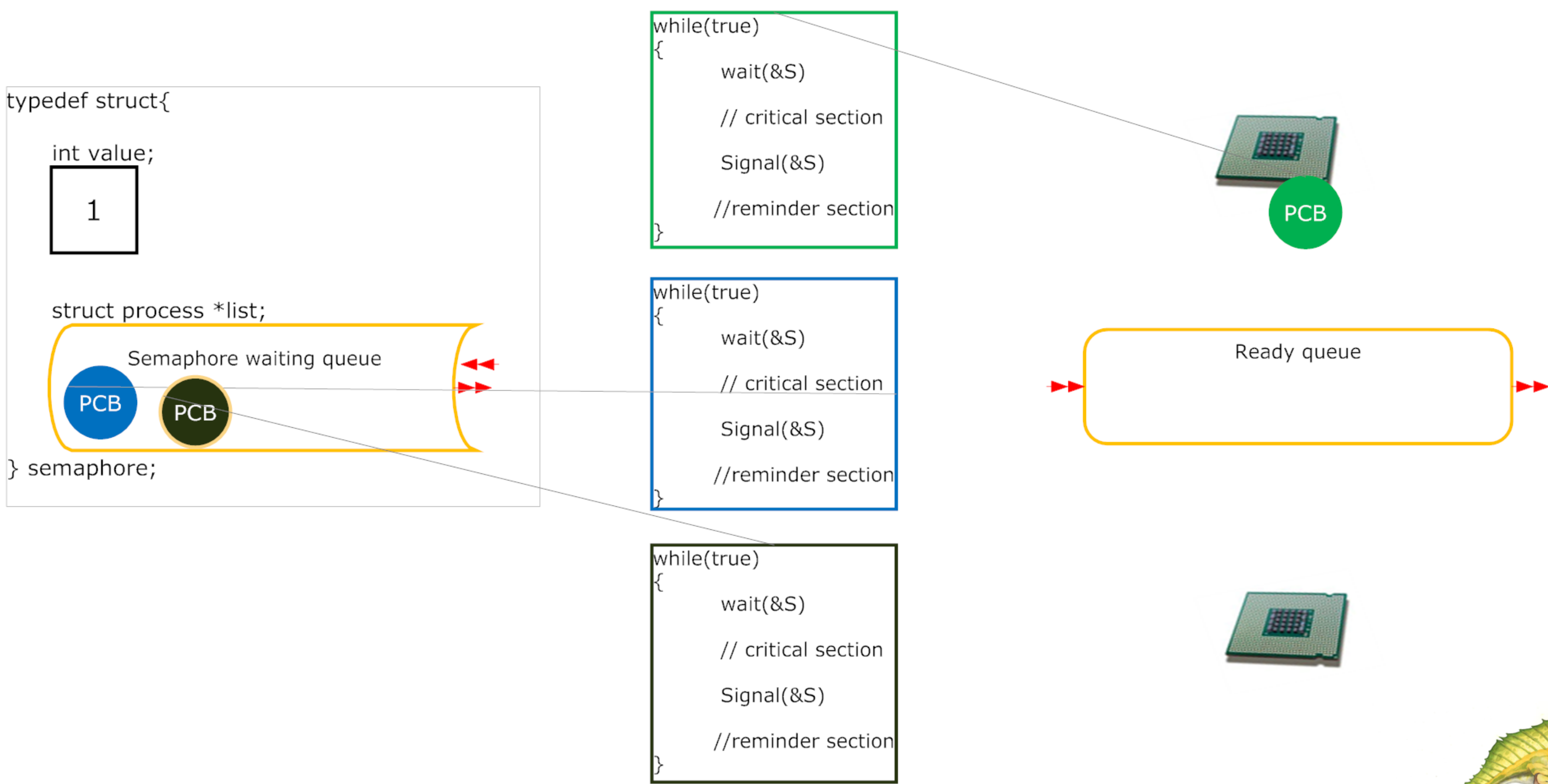
```
wait(S);
wait(Q);
 ...
signal(S);
signal(Q);
```

$P_1$

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

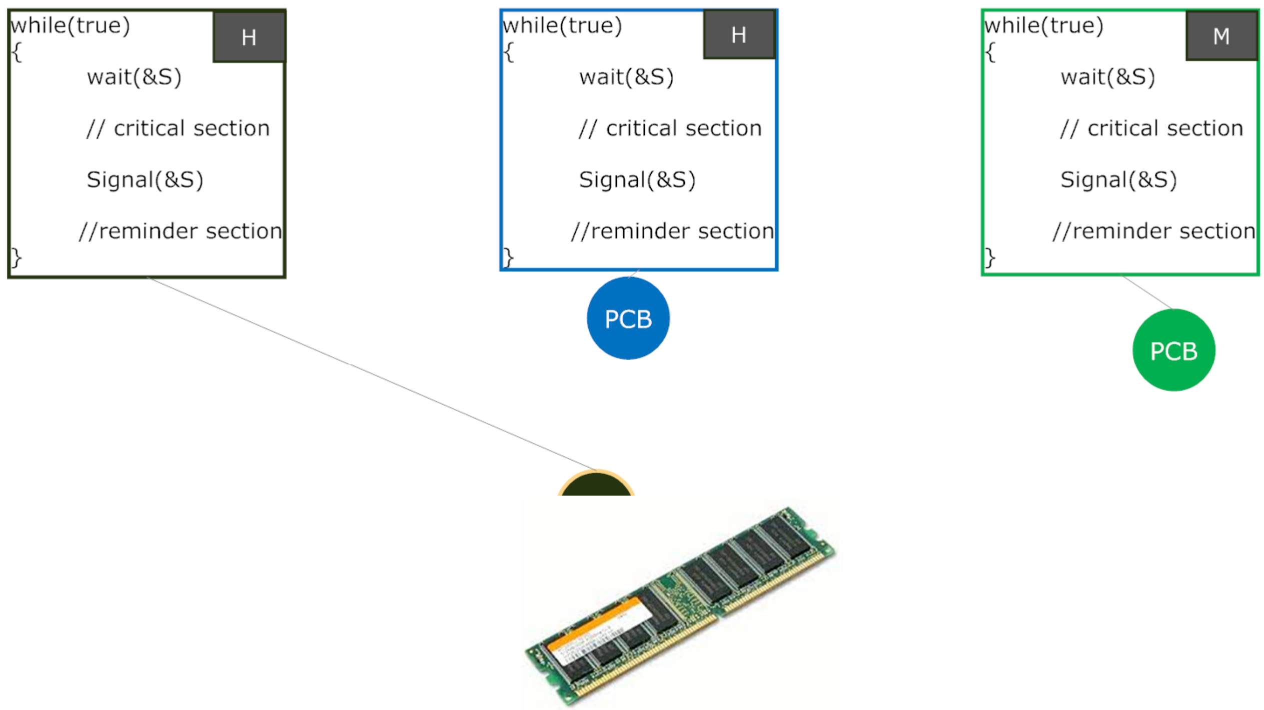
# Starvation

- **Starvation** – **indefinite blocking**
  - A process may never be removed from the semaphore queue in which it is suspended if operate the semaphore list as a stack (LIFO)



# Priority Inversion

- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via **priority-inheritance protocol**



# How to use in general

```
//process
do {



    //critical section (shared resources)



} while (true);
```

while(test_and_set(&lock));

lock=f

acquire (mutex);

release(mutex);

wait(semaphore);

signal(semaphore);

# CLASSICAL PROBLEMS OF SYNCHRONIZATION

# Classical Problems of Synchronization

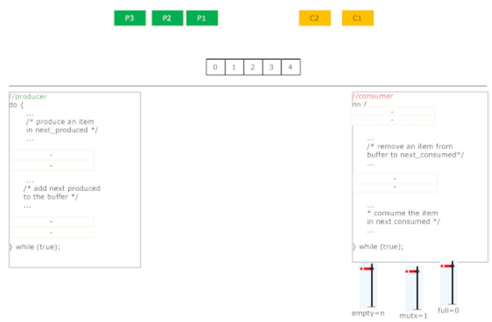
- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

# 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

# Producer-Consumer solution

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



```
//consumer
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);
```

# 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.)

- The structure of the consumer process

# Readers-Writers Problem

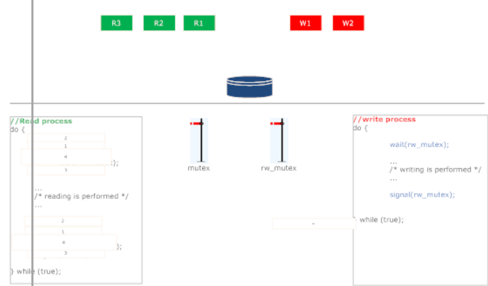
- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do ***not*** perform any updates
  - Writers – can both read and write
- Problem – allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same 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 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

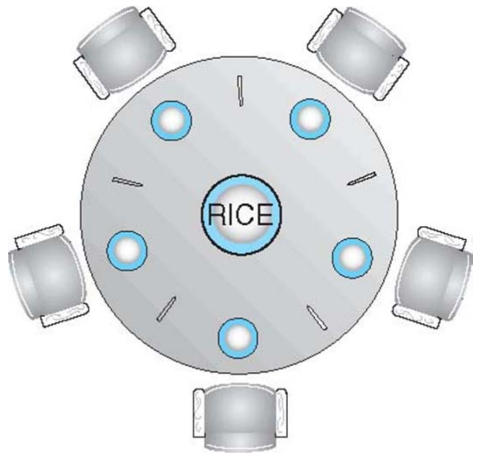
# Solution for first variation

```
//write process
do {
        wait(rw_mutex);

        ...
        /* writing is performed */
        ...

       signal(rw_mutex);

} while (true);
```

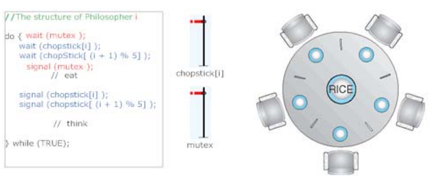



```
//Read process
do {

         wait(mutex);
         read_count++;
         if (read_count == 1)
              wait(rw_mutex);
         signal(mutex);
         ...
         /* reading is performed */
         ...
         wait(mutex);
         read count--;
         if (read_count == 0)
               signal(rw_mutex);
          signal(mutex);
} while (true);
```

# 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.

# Naming convention

| | | |
|---|---|---|
| Mutex locks with busy wait | = | Spinlock |
| Binary semaphore (value=1) with no busy wait (using waiting queue) | = | Mutex locks |
| Counting semaphore (value=n) | = | Semaphore |

# KERNEL LEVEL SYNCRONIZATION EXAMPLES

# Windows Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses **spinlocks** on multiprocessor systems
  - Spinlocking-thread will never be preempted
- Also provides **dispatcher objects** user-land which may act mutexes, semaphores, events, and timers
  - **Events**
    - An event acts much like a condition variable
  - Timers notify one or more thread when time expired
  - Dispatcher objects either **signaled-state** (object available) or **non-signaled state** (thread will block)

# Linux Synchronization

- Linux:
  - Prior to kernel Version 2.6, disables interrupts to implement short critical sections
  - Version 2.6 and later, fully preemptive
- Linux provides:
  - Semaphores
  - atomic integers
  - spinlocks
  - reader-writer versions of both
- On single-cpu system, spinlocks replaced by enabling and disabling kernel preemption

# USER LEVEL SYNCHRONIZATION EXAMPLES

# Windows API mutex

http://sallamah.weebly.com/uploads/6/9/3/5/6935631/winmutex.c

# POSIX (ptread) mutex

http://sallamah.weebly.com/uploads/6/9/3/5/6935631/posixmutex.c

# Pthreads Synchronization

- Pthreads API is OS-independent
- It provides:
  - mutex locks
  - condition variable
- Non-portable extensions include:
  - read-write locks
  - spinlocks

# Alternative Approaches

- Transactional Memory
- OpenMP
- Functional Programming Languages

# Transactional Memory

- A **memory transaction** is a sequence of read-write operations to memory that are performed atomically.

```
void update()
{
     /* read/write memory */
}
```

# OpenMP

- OpenMP is a set of compiler directives and API that support parallel progamming.

```
void update(int value)
 {
         #pragma omp critical
         {
                 count += value
         }
  }
```

The code contained within the **#pragma omp critical** directive is treated as a critical section and performed atomically.

# Functional Programming Languages

- Functional programming languages offer a different paradigm than procedural languages in that they do not maintain state.
- Variables are treated as immutable and cannot change state once they have been assigned a value.
- There is increasing interest in functional languages such as Erlang and Scala for their approach in handling data races.

# End of Chapter 5