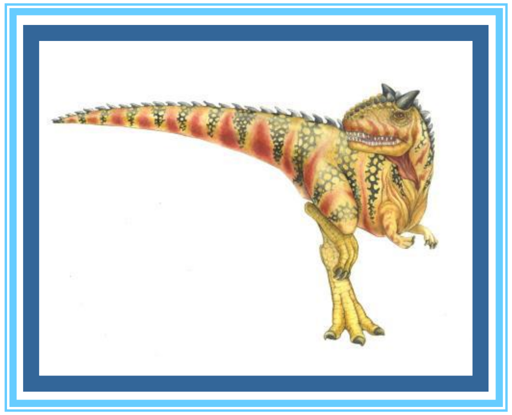
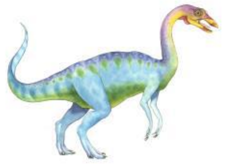
8: Synchronization



Synchronization

- Background
- The Critical-Section Problem
- Peterson’s Solution
- Synchronization Hardware
- Mutex Locks
- Conditional Variables
- Semaphores
- Classic Problems of Synchronization
- Synchronization Examples

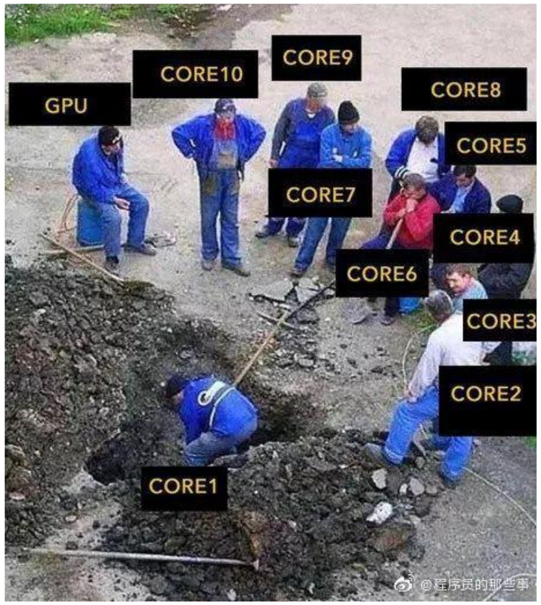


Objectives

- 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

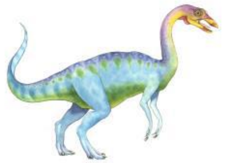
Background

- 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



Background

- 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`** = 0. It is incremented by the producer after it produces a new item to a buffer and is decremented by the consumer after it consumes an item from a buffer.



Producer & Consumer

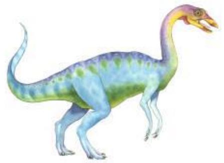
```
item  next_produced;
while (true) {
          /* produce an item in next_produced */
          while (counter == BUFFER_SIZE) //buffer is full
                    ;/* do nothing, such as create data */
          buffer[in] = next_produced;
          in = (in + 1) % BUFFER_SIZE;
          counter++;
}
```

BUFFER_SIZE is the size of the **buffer**

Producer

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

Consumer



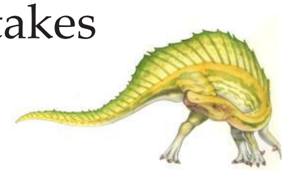
Race Condition

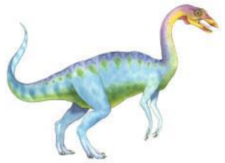
- **counter++** could be implemented as
  register1 = counter
  register1 = register1 + 1
  counter = register1
- **counter--** could be implemented as

  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**}

Race condition: several processes access and manipulate the same data concurrently and the outcome depends on which order each access takes place.



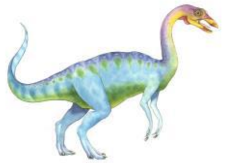


Critical Section Problem

- 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 process is allowed to execute in its critical section
- ***Critical section problem*** is to design a **protocol** that the processes can use to cooperate
- To solve critical section problem, each process must ask permission to enter its critical section in **entry section**, follow critical section with **exit section**, then **remainder section**

(see next slide)

- Especially difficult to solve this problem in preemptive kernels



Critical Section

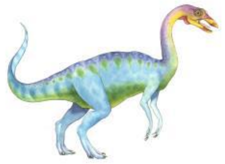
- General structure of process P_i

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

Ask for permission

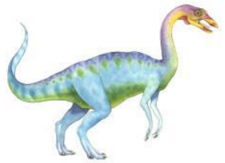
Protect this section

Do sth in order to allow other processes to enter critical section



Solution to Critical-Section Problem

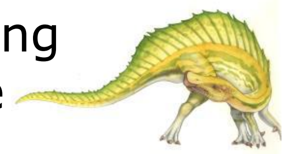
- **A solution to the critical-section problem must satisfy three requirements:**
  1. **Mutual Exclusion** – At any time, only one process can be executing in its critical section
  2. **Progress** - the selection of the processes to enter the critical section next cannot be postponed indefinitely
  3. **Bounded Waiting** - A bound time must exist after a process has made a request to enter its critical section and before that request is granted

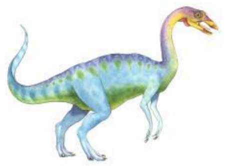


How about disable the interrupts?

```
do {

    entry section        Disable interrupt

        critical section

    exit section         Enable interrupt

        remainder section

} while (true);
```

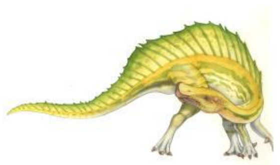
- This can solve single CPU core critical section problem
- This cannot block other processes on other cpu cores
- This cannot prevent processes on multiple CPU cores from entering the critical section at the same time

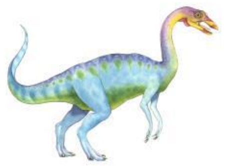




Peterson's Solution

- It is a classic software-based solution to the critical-section problem
- Good algorithmic description of solving the problem
- Solution for **two** processes
- 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!
  - It is initialized to ***false***.





Algorithm for Process P_i & P_j

```
do {
          flag[i] = true; //ready
          turn = j; //allow Pj to enter
          while (flag[j] && turn = = j);
          critical section
          flag[i] = false;
          remainder section
} while (true);
```

Ask for entry permission

P_i

Exit

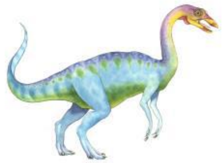
flag[i] and flag[j] are initialized to **false**.

```
do {
          flag[j] = true; //ready
          turn = i; //allow Pi to enter
          while (flag[i] && turn = = i);
          critical section
          flag[j] = false;
          remainder section
} while (true);
```

Ask for entry permission

P_j

Exit



Peterson's Solution (Cont.)

- Provable that the three Critical Section(CS) requirements 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

(refer to textbook for details)

Modern Architecture Example

- Two threads share the data:

```
boolean flag = false;
int x = 0;
```

- Thread 1 performs

```
while (!flag)
  ;
print x
```

- Thread 2 performs

```
x = 100;
flag = true
```

- What is the expected output?

  100

Modern Architecture Example (Cont.)

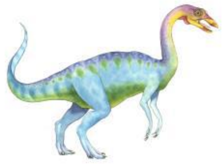
- However, since the variables `flag` and `x` are independent of each other, the instructions:

```
flag = true;
 x = 100;
```

 for Thread 2 may be reordered

- If this occurs, the output may be 0!

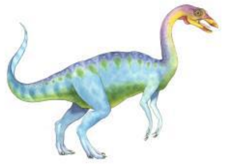
- Need Memory barrier:
  - __sync_synchronize()
  - asm volatile ("" : : : "memory")



Synchronization Hardware

- Software-based solutions (such as Peterson’s) are not guaranteed to work on modern computer architecture
- Many systems provide **hardware support** for implementing the critical section code.
- Based on idea of **locking**
  - Protecting critical regions via locks
- As mentioned earlier, **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

  1. Test memory word and set value
  2. Swap contents of two memory words



Mutual Exclusion

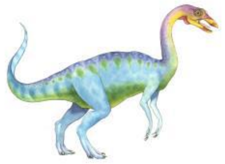
- Fundamental difficulty in implementing mutual exclusion: cannot read/write shared memory at the same time
  - If “load”, writing is not allowed.
    - The information you “load” is instantly outdated.
  - If “store”, reading is not allowed
    - During the blind “store”, you don’t know what you write to and what you are writing.
  - Uncontrolled scheduling / interrupt may happen untimely

- Suppose the hardware can provide us an "instant" read + write instruction
  - Ask everyone to close eyes, take a look (load), and update (store)
  - If multiple requests are made at the same time, choose one “winner”
  - The "loser" has to wait for the "winner" to finish before proceeding

Solution to Critical-section Problem Using Locks

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

- The lock acquisition (entry section) and lock release (exit section) are each done using **a single fast atomic hardware instruction**.
- A process that wants to enter the critical section must **first get the lock.**
- If the lock is already acquired by another process, the process will **wait until the lock becomes available**.



test_and_set Instruction

Definition:

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

1. Executed **atomically** by CPU as a single hardware instruction.
2. The C code here is just to explain how the hardware instruction works.
3. In practice, `target` is a pointer to the lock itself, which is a boolean variable in memory shared by all the processes that want to acquire the lock.
4. Set the new value of the lock to `TRUE`.
5. Returns as result the **original** value of the lock.

Solution using test_and_set()

- Shared Boolean variable **lock**, initialized to FALSE
- Solution using test_and_set:

```
do {
        while (test_and_set(&lock))
             ; /* do nothing */
        /* critical section */
        lock = false;
        /* remainder section */
} while (true);
```

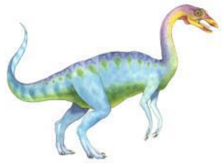
```
do {
        acquire lock
        critical section
        release lock
        remainder section
} while (TRUE);
```

When lock = true, keep while looping.

When lock = false, process can enter the critical section
And set lock = true, don't allow other processes to enter.

After finish the critical section, reset lock = false, to allow other processes to enter the critical section.

Although this algorithm satisfies the mutual-exclusion requirement, it does not satisfy the bounded-waiting requirement.



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** by CPU as a single hardware instruction.
2. The C code here is just to explain how the hardware instruction works.
3. In practice, `value` is a pointer to the lock itself, which is an integer variable in memory shared by all the processes that want to acquire the lock.
4. Set `*value` (the lock) to the value of the passed parameter `new_value` but only if `*value == expected`. That is, the swap takes place only under this condition.
5. Returns as result the **original** value of the lock.
6. Similar to `test_and_set` but with an integer lock and an extra condition.

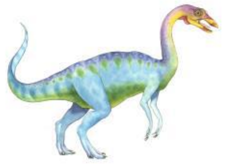
Solution using compare_and_swap

- Shared integer "lock" initialized to **false**;
- Solution using compare_and_swap:

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

       /* critical section */

       lock = false;

       /* remainder section */

} while (true);
```

test_and_set(&lock)

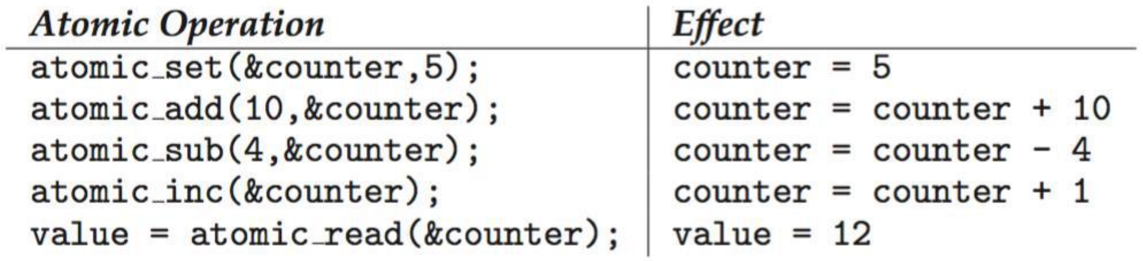
Although this algorithm satisfies the mutual-exclusion requirement, it does not satisfy the bounded-waiting requirement, either.



Atomic

- atomic integers

```
atomic_t counter;
int value;
```

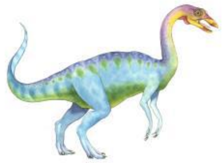


Atomic Operation	*Effect*
`atomic_set(&counter,5);`	`counter = 5`
`atomic_add(10,&counter);`	`counter = counter + 10`
`atomic_sub(4,&counter);`	`counter = counter - 4`
`atomic_inc(&counter);`	`counter = counter + 1`
`value = atomic_read(&counter);`	`value = 12`

- atomic operation implementation

```
int atomic_CAS(volatile int *addr, int expected, int newval) {
        asm volatile ("lock cmpxchg %[new], %[ptr]"
        : "+a"(expected), [ptr] "+m"(*addr)
        : [new] "r"(newval)
        : "memory");
        return expected;
}
```

- Standard library header <stdatomic.h>



Solution using compare_and_swap

- Mutual exclusion: Yes
- Progress: Yes
- Bounded Waiting: No
- Busy-waiting: Yes
- For the exact same reasons as for the solution that uses **`test_and_set`**.
- Therefore the solution presented in the previous slide is not good enough either.

Bounded-waiting Mutual Exclusion with test_and_set

P_i

```
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);
```

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

The common data structures are

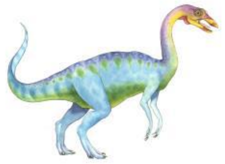
boolean waiting[n];
boolean lock;

These data structures are initialized to **false**.

(refer to the text book for details)

Spinlock issues

- Performance Issues (0)
  - Spinning (shared variables) may trigger cache synchronization among processors, increasing latency
- Performance issues (1)
  - Threads on other processors are idle spinning except the thread entered the critical section
  - The more processors compete for the lock, the lower the utilization of the CPU
- Performance issues (2)
  - The thread that acquired the spinlock may be switched out by the operating system
  - The OS doesn't "sense" what the thread is doing (But why not?)
  - Achieve 100% waste of resources

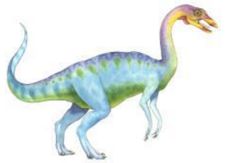


Spinlock usage and sleeplock

1. The critical section is hardly "congested“
2. Execution stream switching is prohibited while holding a spinlock

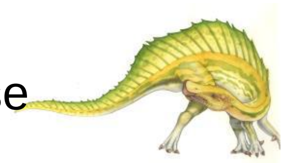
- Usage Scenario: Concurrent Data Structures in OS Kernel (short critical sections)
  - OS can turn off interrupts and preemption
  - Guarantees that the lock holder can release the lock within a short period of time

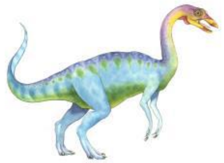
- Implement thread + long critical section mutual exclusion
  - Put the lock implementation into OS
  - Sleep lock: acquire/release via system calls
  - acquire lock
    - If fail, sleep, switch to another thread
  - release lock
    - If there are threads waiting, wake up one



Spinlock usage scenarios

- Spinlock (Threads directly share lock)
  - **Faster** fast path (short critical section)
    - cmpxchg succeed → enter critical section, low cost
  - **Slower** slow path (long critical section)
    - cmpxchg failed → wasteful CPU spinning wait
- Sleeplock (Use system calls to query lock)
  - **Faster** slow path
    - acquire lock failed → won’t occupy CPU
  - **Slower** slow path
    - acquire lock successful → still need to trap to kernel (syscall)
- So how about:
  - Fast path: one atomic instruction, return after acquiring the lock
  - Slow path: acquire lock failed, syscall to sleep
  - Optimization trick: focus on average (frequent) not worst case





POSIX Mutex Locks

```
#include <pthread.h>
pthread_mutex_t mutex;

pthread_mutex_init(&mutex,NULL);
```

pthread_mutex_lock(&mutex); $\longrightarrow$ *acquire lock*

pthread_mutex_unlock(&mutex); $\longrightarrow$ *release lock*

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

int pthread_mutex_init(pthread_mutex_t ****mutex*, const pthread_mutexattr_t ****mutexattr***);**

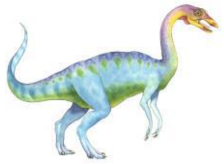
The pthread_mutex_init() function initializes the specified mutex. If *attr* is non-NULL, the attributes specified are used to initialize the mutex. If the attribute object is modified later, the mutex's attributes are not affected. If *attr* is NULL, the mutex is initialized with default attributes, as specified for pthread_mutexattr_init().

A mutex can be statically initialized by assigning PTHREAD_MUTEX_INITIALIZER in its definition, as follows:
pthread_mutex_t def_mutex = PTHREAD_MUTEX_INITIALIZER;

A mutex must be initialized (either by calling pthread_mutex_init(), or statically) before it may be used in any other mutex functions.

Conditional Variables

- Conditional Variables (CV)
  - Use sleep not spin
  - Wake up when the operation is done
  - Often use with mutex
- Condition Variable API
  - wait(cv, mutex) 💤
    - When calling, you must ensure that the mutex has been obtained
    - release mutex, go to sleep
  - signal/notify(cv) 💬 Private message: let's go
    - If there are threads waiting for cv, wake up one of them
  - broadcast/notifyAll(cv) 📢 Everyone: let's go
    - wake up all threads waiting for cv

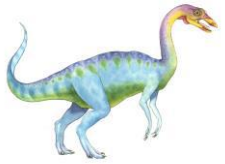


Conditional Variables

```
void Tproduce() {
    mutex_lock(&lk_empty); //mutex ensure the condition is meet
    while (empty_slot == 0)
            cond_wait(&cv_empty, &lk_empty);
    empty_slot--;
    mutex_unlock(&lk_empty);

    addtobuffer();

    mutex_lock(&lk_filled);
    filled_slot++;
    cond_signal(&cv_filled);
    mutex_unlock(&lk_filled);
}
```

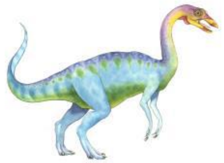
Conditional Variables

```
void Tconsume() {
    mutex_lock(&lk_filled); //mutex ensure the condition is meet
    while (filled_slot == 0)
            cond_wait(&cv_filled, &lk_empty);
    filled_slot--;
    mutex_unlock(&lk_filled);

    removefrombuffer();

    mutex_lock(&lk_empty);
    empty_slot++;
    cond_signal(&cv_empty);
    mutex_unlock(&lk_empty);
}
```



Current jobs

```
struct job {
      void (*run)(void *arg); void *arg;
}
while (1) {
    struct job *job;
    mutex_lock(&mutex);
    while (! (job = get_job()) ) {
      wait(&cv, &mutex);
    }
    mutex_unlock(&mutex);

    job->run(job->arg); //no need to hold the lock
                        //can generate new jobs
                        //release the allocated res
}
```



POSIX Condition Variables

- Since POSIX is typically used in C/C++ and these languages do not provide a monitor, POSIX condition variables are associated with a POSIX mutex lock to provide mutual exclusion: Creating and initializing the condition variable:

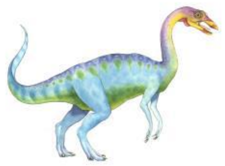
```
pthread_mutex_t mutex;
pthread_cond_t cond_var;

pthread_mutex_init(&mutex,NULL);
pthread_cond_init(&cond_var,NULL);
```

- Thread waiting for the condition **`a == b`** to become true:

```
pthread_mutex_lock(&mutex);
while (a != b)
     pthread_cond_wait(&cond_var, &mutex);

pthread_mutex_unlock(&mutex);
```

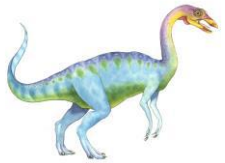
- Thread signaling another thread waiting on the condition variable:

```
pthread_mutex_lock(&mutex);
a = b;
pthread_cond_signal(&cond_var);
pthread_mutex_unlock(&mutex);
```



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 atomic operations**
  - **`wait()`** and **`signal()`**
    - Originally called **`P()`** and **`V()`**



Semaphore

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

```
Semaphore S
```

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

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

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

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

Semaphore Usage

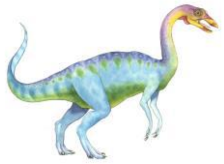
- Counting semaphore – integer value can range over an unrestricted domain
- Binary semaphore – integer value can range only between 0 and 1
  - Same as a **mutex lock**
- Can **solve various synchronization problems**
- Consider two processes ***P_1*** and ***P_2*** that require ***S_1*** to happen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

  Create a semaphore "**synch**" **initialized to 0**

```
semaphore  synch;
synch = 0
```

```
P1:
   S1;
   signal(synch);
```

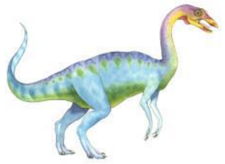
```
P2:
   wait(synch);
   S2;
```



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

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

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



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;
```

Implementation with no Busy waiting (Cont.)

- Again, C code is just for explanation:

Busy waiting

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


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

No busy waiting

```
wait(semaphore *S) {
   S->value--;
   if (S->value < 0) {
      //add this process to S->list;
      block();//put in a waiting queue
   }
}

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

POSIX Named Semaphore

#include <semaphore.h>
sem_t *sem_mutex;

Normally used among different processes

sem_t *sem_open(const char *_name_, int _oflag_, mode_t _mode_, unsigned int _value_);

sem_open("SEM", O_CREATE, 0666, 1);

multiple processes can easily use a common semaphore as a synchronization mechanism by the semaphore’s name.

```
                              do {
sem_wait(sem_mutex);   ---->     acquire lock
                                       critical section
sem_post(sem_mutex);   ---->     release lock
                                       remainder section
                              } while (true);
```

POSIX Unnamed Semaphore

#include <semaphore.h>
sem_t sem_mutex;

Normally used among different threads within a process

sem_init(&sem_mutex,0, 1);

int sem_init(sem_t ***_sem_, int** _pshared_**, unsigned int** _value_**);**

three parameters:
1. A pointer to the semaphore
2. A flag indicating the level of sharing
3. The semaphore's initial value

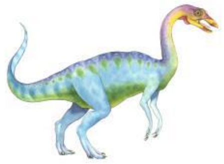
sem_wait(&sem_mutex); → acquire lock

sem_post(&sem_mutex); → release lock

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

Classical Problems of Synchronization

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



Bounded-Buffer Problem

- ***Producer-Consumer problem***
  - ***n*** buffers, each holds 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.)

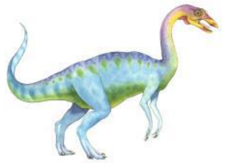
- The structure of the producer process

```
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0
```

The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The empty and full semaphores count the number of empty and full buffers.

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

```
item next_produced;
while (true) {
    /* produce an item in next_produced */
    while (counter == BUFFER_SIZE)
        ;/* do nothing, such as create data */
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

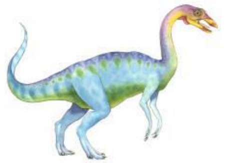


Bounded Buffer Problem (Cont.)

- The 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);
```

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



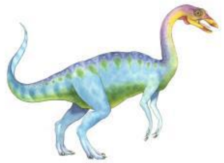
Bounded Buffer Problem (Cont.)

posix_bb.c

Compile command:

```
g++ -o posix_bb posix_bb.c -lpthread
```

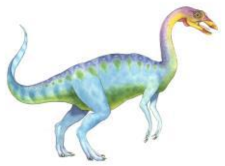
```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <semaphore.h>
#include <deque>
#include <unistd.h>
#include <signal.h>

#define TRUE 1

/* functions for threads to call */
void *producer(void *param);  //for producer
void *consumer(void *param); //for consumer

//using three semaphores
sem_t  empty;
sem_t  full;
sem_t  mutex;

int shared_item = 0;

std::deque<int> myboundedbuffer;
pid_t pid;
```



Bounded Buffer Problem (Cont.)

posix_bb.c

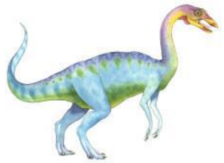
```
int main(int argc, char *argv[]) {
            int i, scope;
            pthread_t producerID, consumerID;      /* the thread identifier */
            pthread_attr_t attr;                   /* set of attributes for the thread */
            int n = 10;
            pid = getpid();

            //create semaphores, and initialize
            sem_init(&empty, 0, n);//n
            sem_init(&full, 0, 0);//0
            sem_init(&mutex, 0, 1);//1

            /* get the default attributes */
            pthread_attr_init(&attr);

            /* create the threads */
            pthread_create(&producerID, &attr, producer, &producerID);
            pthread_create(&consumerID, &attr, consumer, &consumerID);

            /*Now join on each thread*/
            pthread_join(producerID, NULL);
            pthread_join(consumerID, NULL);
            return 0;
}
```



Bounded Buffer Problem (Cont.)

posix_bb.c

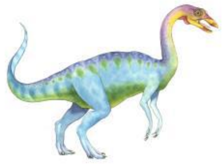
```
/*The thread will begin control in this function.*/
void *producer(void *param) {
        int id = *(int*)param;
        printf("producer Thread ID = %d\n", id);

        do {
                sem_wait(&empty);
                sem_wait(&mutex);

                shared_item++;
                if (shared_item > 20)
                        break;

                printf("Producer create an item %d\n", shared_item)
                myboundedbuffer.push_back(shared_item);

                sem_post(&mutex);
                sem_post(&full);
        }while (TRUE);

        kill(pid, SIGINT); //send a signal to kernel to terminate the process
        pthread_exit(0);
}
```



Bounded Buffer Problem (Cont.)

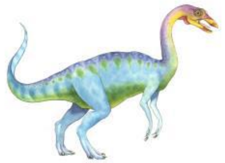
posix_bb.c

```
void *consumer(void *param) {

        int id = *(int*)param;
        printf("consumer Thread ID = %d\n", id);

        do {
                sem_wait(&full);
                sem_wait(&mutex);

                printf("\tConsumer process an item %d\n",
                        myboundedbuffer.front()) ;
                myboundedbuffer.pop_front();

                sem_post(&mutex);
                sem_post(&empty);

        } while (TRUE);

        pthread_exit(0);
}
```



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 data
- 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



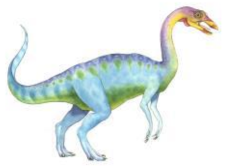
Readers-Writers Problem (Cont.)

- The structure of a writer process

The **mutex** semaphore is used to ensure mutual exclusion when the variable read_count is updated.
The **read_count** variable keeps track of how many processes are currently reading the object.
The semaphore **rw_mutex** functions as a mutual exclusion semaphore for the writers. It is also used by the first or last reader that enters or exits the critical section.

semaphore rw_mutex = 1;
semaphore mutex = 1;
int read_count = 0;

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



Readers-Writers Problem (Cont.)

- The structure of a reader 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);
```

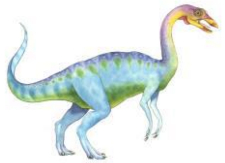
1st reader process

last reader process

Readers-Writers Problem (Cont.)

posix_rw.c
Compile command:
gcc -o posix_rw posix_rw.c -lpthread

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <semaphore.h>
#include <unistd.h>
#include <signal.h>

#define TRUE 1

/* the thread runs in this function */
void *writer(void *param);
void *reader(void *param);

sem_t  rw_sem;
sem_t  mutex;
int read_count = 0;

int shared_data = 0;
pid_t pid;
```



Readers-Writers Problem (Cont.)

posix_rw.c

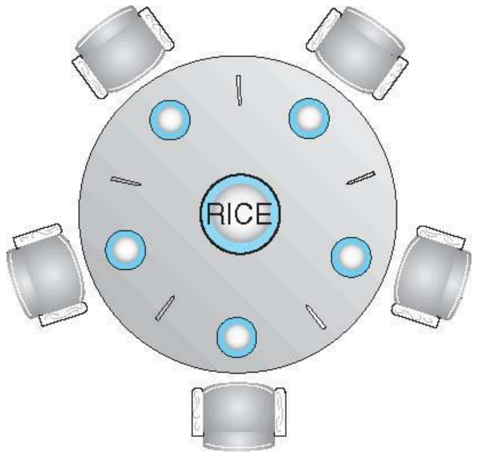
```
int main(int argc, char *argv[]) {
          int i, scope;
          pthread_t writerID, readerID1, readerID2;        /* the thread identifier */
          pthread_attr_t attr;              /* set of attributes for the thread */

          pid = getpid();

          //create semaphores, and initialize to 1
          sem_init(&rw_sem, 0, 1);
          sem_init(&mutex, 0, 1);

          /* get the default attributes */
          pthread_attr_init(&attr);

          /* create 3 threads */
          pthread_create(&writerID, &attr, writer, &writerID);
          pthread_create(&readerID1, &attr, reader, &readerID1);
          pthread_create(&readerID2, &attr, reader, &readerID2);

          /* Now join on each thread */
          pthread_join(writerID, NULL);
          pthread_join(readerID1, NULL);
          pthread_join(readerID2, NULL);
          return 0;
}
```

Readers-Writers Problem (Cont.)

posix_rw.c

```
void *writer(void *param) {
        int id = *(int*)param;
        printf("Writer Thread ID = %d\n", id);
        shared_data = 0;
        do {
                sem_wait(&rw_sem);
                shared_data++;
                printf("Writer:  new data = %d\n", shared_data);
                sem_post(&rw_sem);
                sleep(1);
        } while (TRUE);

        pthread_exit(0);
}
```

Dining-Philosophers Problem

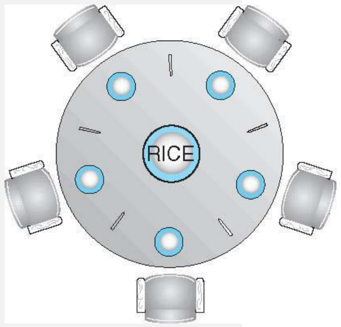


- Philosophers spend their lives alternating **thinking** and **eating**
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both chopsticks to eat, then release both when done
- In the case of 5 philosophers
  - Shared data
    - **Bowl of rice (**data set)
    - **Semaphore** chopstick [5] initialized to 1

Dining-Philosophers Problem Algorithm

- The structure of Philosopher **i**:

```
do {
        wait (chopstick[i] );                  //left chopstick
        wait (chopstick[ (i + 1) % 5] );   //right chopstick
        //  eat state
        signal (chopstick[i] );
        signal (chopstick[ (i + 1) % 5] );
        //  think state
} while (TRUE);
```



- What is the problem with this algorithm?

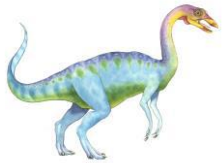
Although this solution guarantees that no two neighbors are eating simultaneously, but it could create a **deadlock**.

Suppose that all five philosophers become hungry at the same time and each grabs the left chopstick. No chopstick left. When each philosopher tries to grab the right chopstick, he/she will be delayed forever.

Dining-Philosophers Problem Algorithm (Cont.)

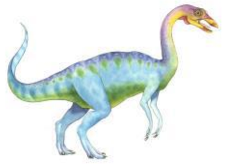
- **Deadlock handling**
  1. Allow **at most 4** philosophers to be sitting simultaneously at the table.
  2. Allow a philosopher to pick up the chopsticks **only if both are available** (picking must be done in a critical section.)
  3. **Use an asymmetric solution** -- an odd-numbered philosopher picks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

```
mutex_lock(&mutex);
while (!(avail[lhs] && avail[rhs])) {
        wait(&cv, &mutex); }
avail[lhs] = avail[rhs] = false;
mutex_unlock(&mutex);

mutex_lock(&mutex);
avail[lhs] = avail[rhs] = true;
broadcast(&cv);
mutex_unlock(&mutex);
```



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)
  - It is the responsibility of the application programmer to use semaphores correctly.
- Deadlock and starvation are possible.



Differences

- Mutex vs Binary Semaphore
  - Mutex has the concept of owner
  - Mutex lock/unlock is done using the same thread
    - Semaphore can wait/signal in different thread

- Conditional variable vs Semaphore
  - Very similar
  - Semaphore can be implemented using cv + mutex + counter
    - Conceptually, semaphore provide a more high level abstraction
  - Semaphore is more useful if the problem definition/formulation has a clear "**one unit** of resource"
    - E.g. one unit of buffer space

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

  (**Most OSes do not prevent or deal with deadlocks**)

- Let ***S*** and ***Q*** be two semaphores initialized to 1

```
        P0                              P1
    wait(S);                        wait(Q);
    wait(Q);                        wait(S);
     ...                             ...
    signal(Q);                      signal(S);
    signal(S);                      signal(Q);
```

- Can cause **starvation** and **priority inversion**
  - **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

Deadlock and Starvation

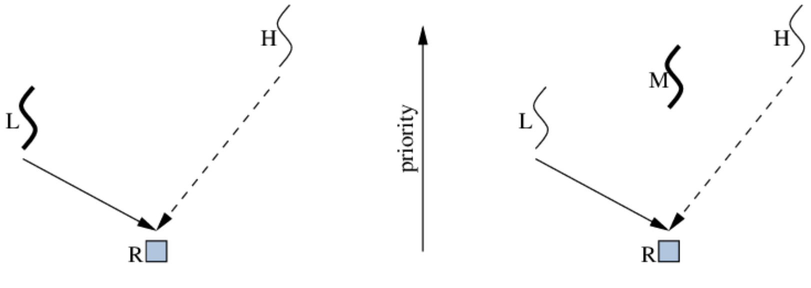
- **priority inversion example:**

Assume we have three processes—*L*, *M*, and *H* —the priority order: $L < M < H$.

Assume that process *H* requires resource *R*, which is currently hold by process *L*. Process *H* would wait for *L* to finish using resource *R*.

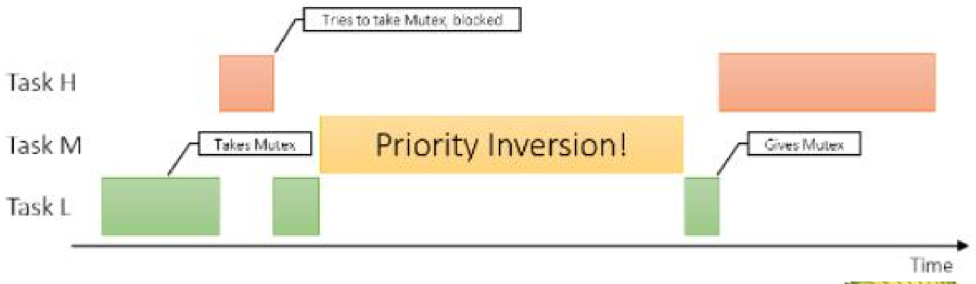
However, process *M* becomes runnable, and preempts process *L*.

Indirectly, a process with a lower priority—process *M*—has affected how long process *H* must wait for *L* to relinquish resource *R*.



priority inversion

unbounded priority inversion



Deadlock and Starvation

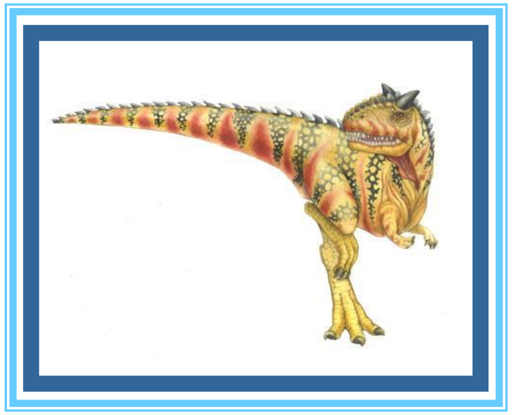
Solution via priority-inheritance protocol

All processes that are accessing resources needed by a higher-priority process inherit the higher priority until they are finished with the resources in question.

When they are finished, their priorities revert to their original values.

This protocol solve the previous **priority inversion** problem.

End of Lecture 8



Sleeping barber problem

- A barbershop has N chairs and one barber
  - The barber does one of two things
  - If there is at least one customer, he chooses one and cuts his hair
  - If there are no customers, he goes to sleep (waits for a customer)
- When a customer enters
  - If there is at least one seat available, he seats himself and tells the barber he would like a haircut, and waits for a haircut
  - If there are no seats available, he leaves

Sleeping barber problem

```
Semaphore barbers = 0;
Semaphore mutex = 1;
Semaphore customers = 0;
int nFreeWRSeats = N;

void Barber(){
     while (1) {
          wait(customers);
          wait(mutex);
          nFreeWRSeats ++;
          signal(barbers);
          signal(mutex);
          haircut();
     }
}
```

```
void Customer() {
    wait(mutex);
    if (nFreeWRSeats > 0) {
      nFreeWRSeats --;
      signal(customers);
      signal(mutex);
      wait(barber);
      get_haircut();
    }
    else {
      signal(mutex);
      //leaveWR();
    }

}
```