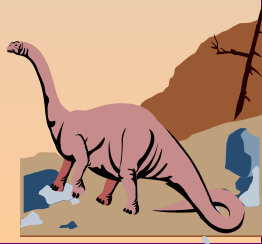
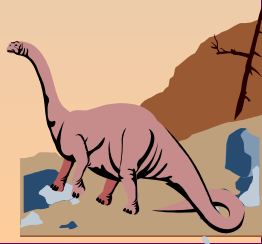
Chapter 6: Process Synchronization

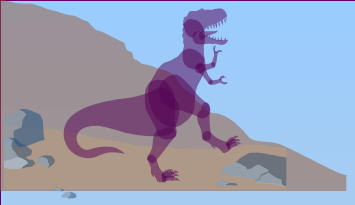
- Background
- The Critical-Section Problem
- Synchronization Hardware
- Semaphores
- Classical Problems of Synchronization



Background

- Producer Consumer Problem
- Solution 1 – circular array
- Solution 2- Suppose that we modify the producer-consumer code by adding a variable counter, initialized to 0 and incremented each time a new item is added to the buffer.

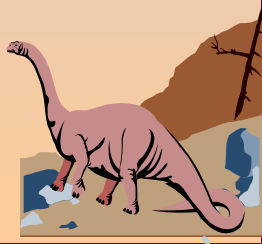


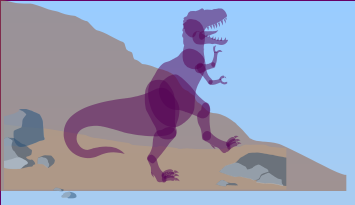


Bounded-Buffer

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
    . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int counter = 0;
```



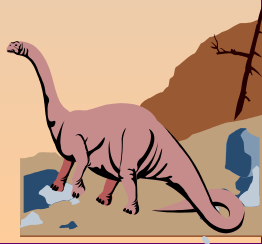


Bounded-Buffer

- Producer process

```
item nextProduced;

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

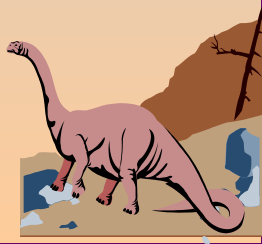


Bounded-Buffer

- Consumer process

```
item nextConsumed;

while (1) {
    while (counter == 0)
        ; /* do nothing */
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
}
```



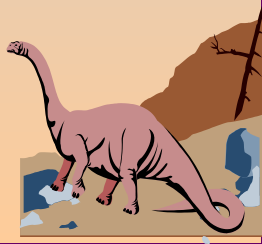
Bounded Buffer

- The statements

  counter++;
  counter--;

  must be performed *atomically*.

- Atomic operation means an operation that completes in its entirety without interruption.



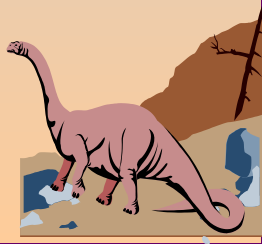
Bounded Buffer

- The statement "**counter++**" may be implemented in machine language as:

  register1 = counter
  register1 = register1 + 1
  counter = register1

- The statement "**counter—**" may be implemented as:

  register2 = counter
  register2 = register2 – 1
  counter = register2

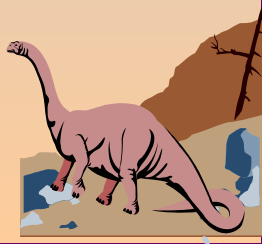


Bounded Buffer

- Assume **counter** is initially 5. One interleaving of statements is:

  producer: **register1 = counter** (*register1 = 5*)
  producer: **register1 = register1 + 1** (*register1 = 6*)
  consumer: **register2 = counter** (*register2 = 5*)
  consumer: **register2 = register2 – 1** (*register2 = 4*)
  producer: **counter = register1** (*counter = 6*)
  consumer: **counter = register2** (*counter = 4*)

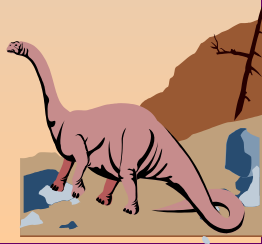
- The value of **counter** may be either 4 or 6, where the correct result should be 5.



Race Condition

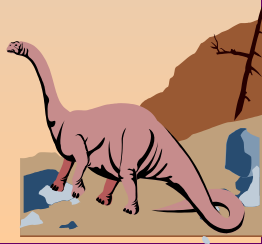
- **Race condition**: The situation where several processes access – and manipulate shared data concurrently. The final value of the shared data depends upon which process finishes last.

- To prevent race conditions, concurrent processes must be **synchronized**.



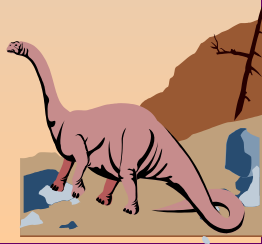
The Critical-Section Problem

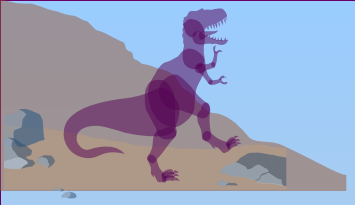
- *n* processes all competing to use some shared data
- Each process has a code segment, called *critical section*, in which the shared data is accessed.
- Problem – ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.



Critical Section Problem

- Consider system of ***n*** processes $\{p_0, p_1, \ldots p_{n-1}\}$
- 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**



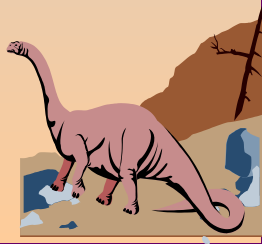


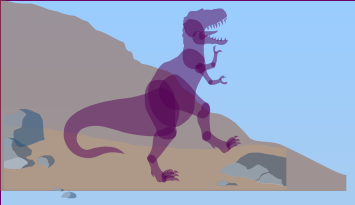
Solution to Critical-Section Problem

A solution to critical section problem must satisfy the following requirements:

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 only those processes that are not executing in their remainder section can participate in the decision on which process will enter the critical section.

   And this selection 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.

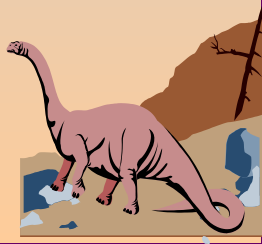


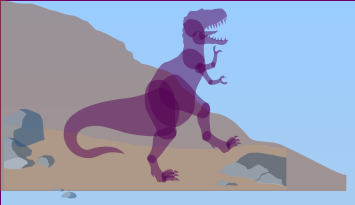


Critical-Section Handling in OS

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



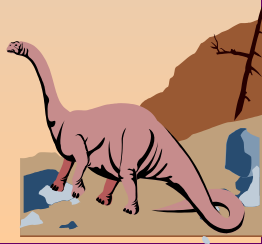


Initial Attempts to Solve Problem

- Only 2 processes, P_0 and P_1
- General structure of process P_i (other process P_j)

```
do {
        entry section
                critical section
        exit section
                reminder section
} while (1);
```

- Processes may share some common variables to synchronize their actions.

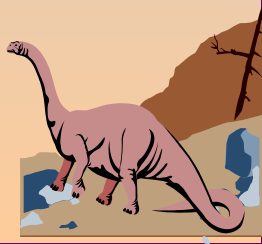


Algorithm 1

- Shared variables:
  - **int turn**;
    initially **turn = 0**
- Process Pi (other process is Pj)

```
do {
while (turn != i) ;
    critical section
turn = j;
    reminder section
} while (1);
```

- .

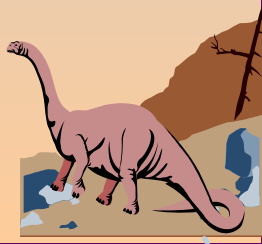


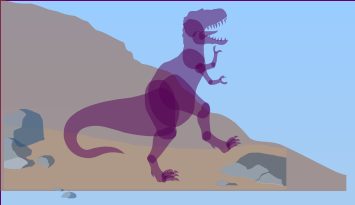
Algorithm 2

- Shared variables
  - **boolean flag[2];**
    initially **flag [0] = flag [1] = false.**
  - **flag [i] = true** ⇒ Pi is ready to enter its critical section.
- Process Pi

```
do {
    flag[i] = true;
    while (flag[j])
        ;
    flag [i] = false;
        remainder section
} while (1);
```

critical section

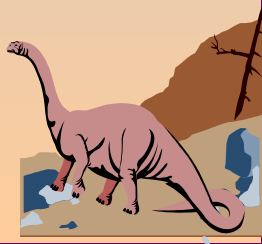


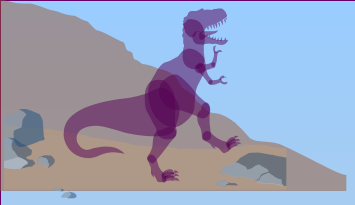


Algorithm 2(contd.)

- Algorithm 2 satisfies mutual exclusion, but not progress requirement.

- T0: P0 sets flag[0] = true
- T1: P1 sets flag[1] = true
- Now both P0 and P1 are looping forever in their respective while loops.





Algorithm 3

- Combined shared variables of algorithms 1 and 2.
- Process P_i

```
do {
        flag [i]= true;
    turn = j;
    while (flag [j] and turn = = j)
                ;
        critical section
    flag [i] = false;
        remainder section
}while (1);
```

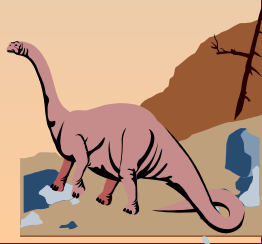
- Meets all three requirements; solves the critical-section problem for two processes.
- This is called Peterson's Solution, software based solution to the critical section problem.

Semaphores

- Critical-section solutions as presented before are difficult to formulate for complex problems.
- Synchronization tool is available called **Semaphore**.
- Semaphore S – integer variable.
- can only be accessed via two indivisible(atomic) operations

```
wait (S):
    while S≤ 0
        ;  //do no-op
S--;

signal (S):
    S++;
```



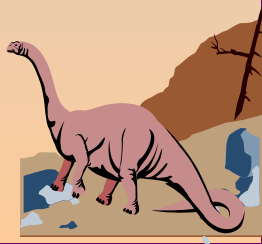
Critical Section of *n* Processes

- Shared data:

  semaphore mutex; //initially mutex = 1

- Process Pi:

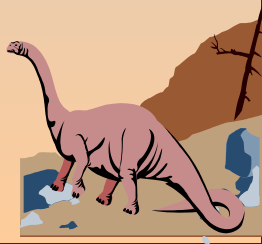
```
do {
        wait(mutex);
            critical section
        signal(mutex);
            remainder section
} while (1);
```



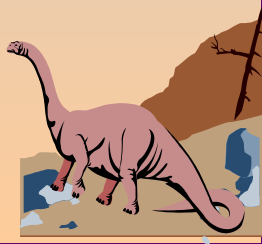
Example of semaphore usage

- Process P1 has to execute instruction S1 and Process P2 has to execute instruction S2
- Suppose we want to execute S2 after S1.
- We do this by P1 and P2 sharing a semaphore “synch”, initialized to 0.

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



- The main disadvantage of solutions before and also with semaphore is **busy waiting.** When a process keeps looping in while loop it is wasting CPU time.
- Semaphores that do busy waiting are called **spinlock** (because the process spins while waiting for a lock)
- In wait() if the value of the semaphore is <=0, the process blocks itself and thus is placed in the waiting queue associated with every semaphore and the process's state is changed to waiting state.
- In signal operation, the blocked process is restarted using a wakeup() operation. The process's state is changed to ready and placed in the ready queue.



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 P_1

  wait(S); wait(Q);

  wait(Q); wait(S);

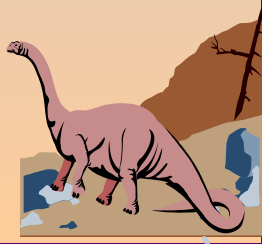
  □ □

  signal(S); signal(Q);

  signal(Q) signal(S);

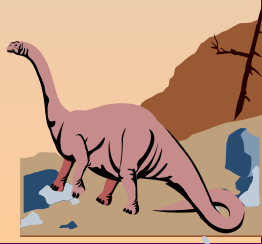
- **Starvation** – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

  This problem occurs when waiting queue of semaphore is implemented as a LIFO order.



Two Types of Semaphores

- Counting semaphore – integer value can range over an unrestricted domain.
- Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement.
- We can implement a counting semaphore S as a binary semaphore.



Implementing *S* as a Binary Semaphore

- Data structures:

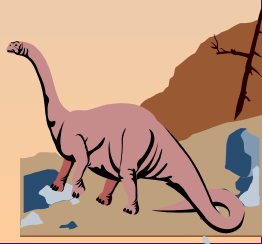
  binary-semaphore S1, S2;

  int C:

- Initialization:

  S1 = 1

  S2 = 0



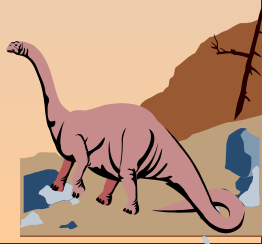
Implementing S

- wait operation

```
wait(S1);
C--;
if (C < 0) {
        signal(S1);
        wait(S2);
        }
signal(S1);
```

- signal operation

```
wait(S1);
C ++;
if (C <= 0)
    signal(S2);
else
    signal(S1);
```



Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem