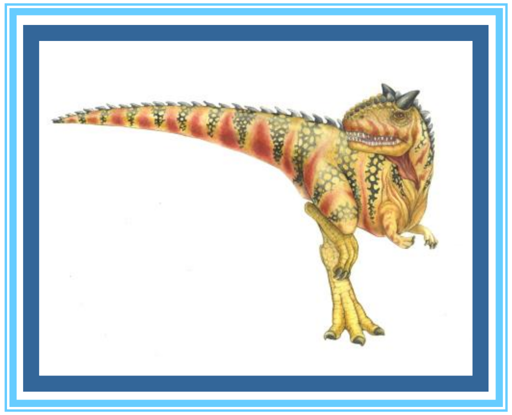
Chapter 6: Synchronization Tools



Too much milk?

	Alice	Bob
12:30	Look in fridge. Out of milk.	
12:35	Leave for store.	
12:40	Arrive at store.	Look in fridge. Out of milk.
12:45	Buy milk.	Leave for store.
12:50	Arrive home, put milk away.	Arrive at store.
12:55		Buy milk.
1:00		Arrive home, put milk away. Oh no!

`x` is a global variable initialized to 0

Thread 1

```
void foo()
{
    x++;
}
```

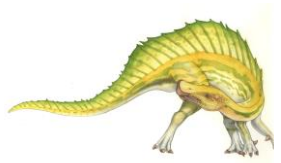
Thread 2

```
void bar()
{
    x--;
}
```

- **After thread 1 and thread 2 finishes, what is the value of x?**
  - could be 0, 1, -1
  - Why?

Synchronization Motivation

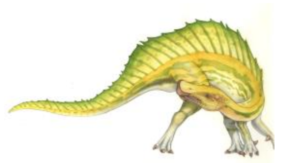
- **Threads cooperate in multithreaded programs**
  - To share resources, access shared data structures
  - To coordinate their execution
- **For correctness, we need to control this cooperation**
  - Thread schedule is non-deterministic
    - Scheduling is not under program control
    - Threads interleave executions arbitrarily and at different rates
    - Behavior changes when re-run program
  - Multi-word operations are not atomic
  - Compiler/hardware instruction reordering



Shared Resources

We initially focus on coordinating access to shared resources

- **Basic problem**
  - If two concurrent threads (processes) are accessing a shared variable, and that variable is read/modified/written by those threads, then access to the variable must be controlled to avoid erroneous behavior
- **Over the next couple of lectures, we will look at**
  - Mechanisms to control access to shared resources
    - Locks, mutexes, sémaphores, monitors, condition variables, etc.
  - Patterns for coordinating accesses to shared resources
    - Bounded buffer, producer-consumer, etc.



Outline

- Background
- The Critical-Section Problem
- Peterson's Solution
- Hardware Support for Synchronization
- Mutex Locks
- Semaphores
- Monitors
- Liveness
- Evaluation

Objectives

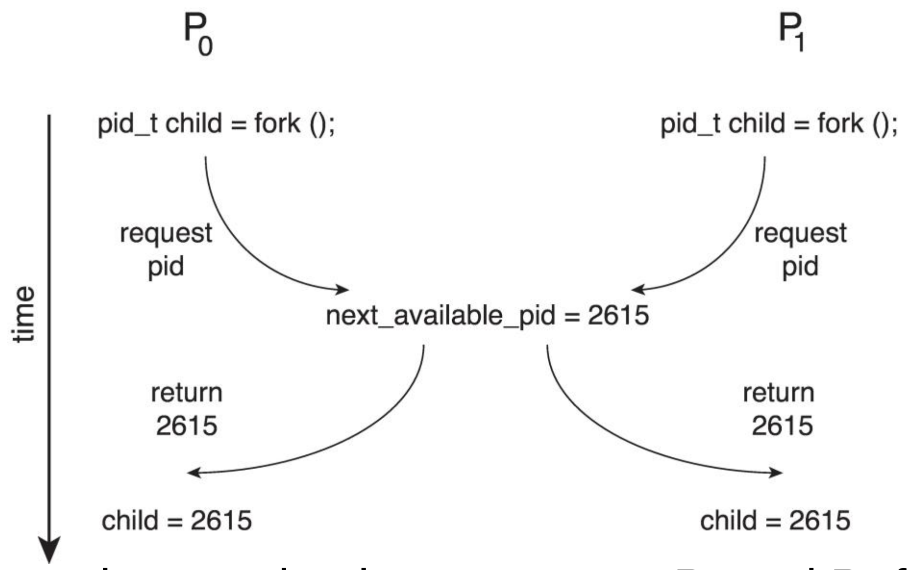
- Describe the critical-section problem and illustrate a race condition
- Illustrate hardware solutions to the critical-section problem using memory barriers, compare-and-swap operations, and atomic variables
- Demonstrate how mutex locks, semaphores, monitors, and condition variables can be used to solve the critical section problem
- Evaluate tools that solve the critical-section problem in low-, Moderate-, and high-contention scenarios

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
- We illustrated in chapter 4 the problem when we considered the Bounded Buffer problem with use of a counter that is updated concurrently by the producer and consumer,. Which lead to race condition.

Race Condition

- Processes P_0 and P_1 are creating child processes using the `fork()` system call
- Race condition on kernel variable `next_available_pid` which represents the next available process identifier (pid)



- Unless there is a mechanism to prevent P_0 and P_1 from accessing the variable `next_available_pid` the same pid could be assigned to two different processes!

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***P_i

```
while (true) {

    entry section

        critical section

    exit section

        remainder section

}
```

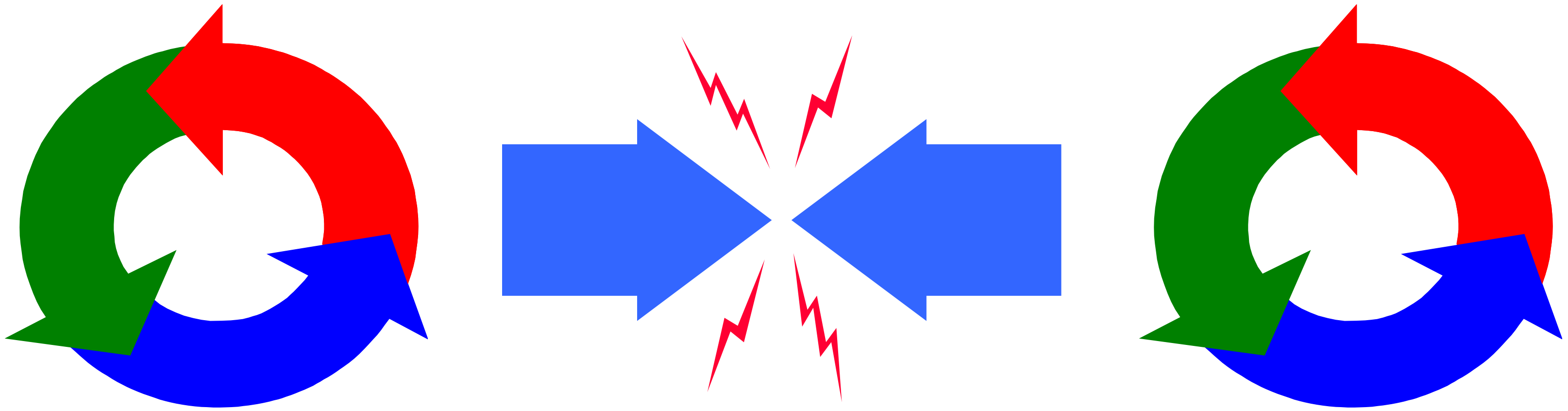
Critical-Section Problem (Cont.)

Requirements for solution to critical-section problem

1. **Mutual Exclusion** - If process $\boldsymbol{P_i}$ 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 process 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

HOW CAN WE SOLVE THIS?

The Mutual Exclusion Problem



The Mutual Exclusion Problem

Eliminating undesirable interleavings is called the mutual exclusion problem.

We need to identify critical sections that only one thread at a time can enter.

We need to devise a pre-protocol and a post-protocol to keep two or more threads from being in their critical sections at the same time.

```
while (true) {
  nonCriticalSection;
  preProtocol;
  criticalSection;
  postProtocol;
}
```

The Mutual Exclusion Problem for N processes

- N processes are executing, in an infinite loop, a sequence of instructions, which can be divided into two sub-sequences: the critical section and the non-critical section. The program must satisfy the mutual exclusion property: instructions from the critical sections of two or more processes must not be interleaved.
- The solution is described by inserting into the loop additional instructions that are to be executed by a process wishing to enter and leave its critical section – the pre-protocol and post-protocol, respectively. These protocols may require additional variables.
- A process may halt in its non-critical section. It may not halt during execution of its protocols or critical section. If one process halts in its non-critical section, it must not interfere with the operation of other processes.

The Mutual Exclusion Problem for N processes

- The program must not deadlock. If some processes are trying to enter their critical sections then one of them must eventually succeed. The program is deadlocked if no process ever succeeds in making the transition from pre-protocol to critical section.
- There must be no starvation of any of the processes. If a process indicates its intention to enter its critical section by commencing execution of the pre-protocol, then eventually it must succeed.
- In the absence of contention for the critical section a single process wishing to enter its critical section will succeed. A good solution will have minimal overhead in this case.

The Mutual Exclusion Problem for 2 processes

We will solve the mutual exclusion problem for two processes using Load and Store to common memory as the only atomic instructions.

```
c1=1;
if (c1==1)
c2=c1;
```

Atomic instructions

One solution to the mutual exclusion problem for two processes is called Dekker's algorithm. We will develop this algorithm in step-by-step sequence of incorrect algorithms: each will demonstrate some pathological behaviour that is typical of concurrent algorithms.

First Attempt

```
int turn=1;
```

```
process P1
while (true) {
 nonCriticalSection1;
 while (turn == 2) {}
 criticalSection1;
 turn = 2;
}
end P1;
```

```
process P2
while (true) {
 nonCriticalSection2;
 while (turn == 1) {}
 criticalSection2;
 turn = 1;
}
end P2;
```

A single shared variable `turn` indicates whose turn it is to enter the critical section.

First Attempt

Mutual exclusion	No deadlock	No starvation	No starvation in absence of contention
✓	✓	✓	✗

Mutual exclusion is satisfied

Proof: Suppose that at some point both processes are in their critical sections. Without loss of generality, assume that P1 entered at time t1, and that P2 entered at time t2, where t1 < t2. P1 remained in its critical section during the interval from t1 to t2.

At time t1, `turn==1`, and at time t2 `turn==2`. But during the interval t1 to t2 P1 remained in its critical section and did not execute its post-protocol which is the only means of assigning `2` to `turn`. At t2 `turn` must still be `1`, contradicting the previous statement.

First Attempt

The solution cannot deadlock
Proof: For the program to deadlock each process must execute the test on `turn` infinitely often failing each time. Therefore, in P1 `turn==2` and in P2 `turn==1`, which is impossible.

There is no starvation
Proof: For starvation to exist one process must enter its critical section infinitely often, while the other executes its pre-protocol forever without progressing to its critical section.
But if P1 executes its even once, it will set `turn==2` in its post-protocol allowing P2 to enter its critical section.

First Attempt

There is starvation in the absence of contention
Proof: Suppose that P2 halts in its non-critical section: `turn` will never be changed from `2` to `1` . P1 may enter its critical section at most one more time. Once P1 sets `turn` to `2`, it will never again be able to progress from its pre-protocol to its critical section.

Even if both processes are guaranteed not to halt this solution must be rejected. Why ?

Second Attempt

```
int c1=1;
int c2=1;
```

```
process P1
while (true) {
 nonCriticalSection1;
 while (c2!=1) {}
 c1=0;
 criticalSection1;
 c1=1;
}
end P1;
```

```
process P2
while (true) {
 nonCriticalSection2;
 while (c1!=1) {}
 c2=0;
 criticalSection2;
 c2=1;
}
end P2;
```

Each process Pi now has its own variable `ci`. Shared variable `ci==0` signals that Pi is about to enter its critical section.

Second Attempt

Mutual exclusion
✘

Mutual exclusion is not satisfied
Proof: Consider the following interleaving beginning with the initial state.

1. P1 checks `c2` and finds `c2==1`.
2. P2 checks `c1` and finds `c1==1`.
3. P1 sets `c1` to `0`.
4. P2 sets `c2` to `0`.
5. P1 enters its critical section.
6. P2 enters its critical section.

Third Attempt

```
int c1=1;
int c2=1;
```

```
process P1
while (true) {
 nonCriticalSection1;
 c1=0;
 while (c2!=1) {}
 criticalSection1;
 c1=1;
}
end P1;
```

```
process P2
while (true) {
 nonCriticalSection2;
 c2=0;
 while (c1!=1) {}
 criticalSection2;
 c2=1;
}
end P2;
```

In Attempt 2 the assignment `Ci=0` was effectively located in the critical section. Try moving it to the beginning of the pre-protocol, `Ci==0` now signals that Pi wishes to enter its critical section.

Third Attempt

Mutual exclusion	No deadlock
✓	✗

Mutual exclusion is satisfied

Proof: Suppose that at some point both processes are in their critical sections. Without loss of generality, assume that P1 entered at time t1, and that P2 entered at time t2, where t1 < t2. P1 remained in its critical section during the interval from t1 to t2.

At time t1, `c1==0` and `c2==1` and at time t2 `c2==0` and `c1==1` . But during the interval t1 to t2 P1 remained in its critical section and did not execute its post-protocol which is the only means of assigning `1` to `c1`. At t2 `c1` must still be `0`, contradicting the previous statement.

Third Attempt

The program can deadlock

Proof: Consider the following interleaving beginning with the initial state.

1. P1 sets `c1` to `0`.
2. P2 sets `c2` to `0`.
3. P1 tests `c2` and remains in the loop.
4. P2 tests `c1` and remains in the loop.

Both processes are locked forever in their pre-protocols.

A Correct Solution: Dekker's Algorithm

- It avoids the strict alternation of a naïve turn-taking algorithm, and was one of the first mutual exclusion algorithms to be invented.
- If both processes attempt to access the critical section at the same time, the algorithm will choose the process whose turn it is. If the other process is already modifying the critical section, it will wait for the process to finish

Dekker's Algorithm – Prove that this algorithm provides Mutual Exclusion and is free from Starvation & deadlock and progresses in the absence of contention

```
int c1=1;
int c2=1;
int turn=1;
```

```
process P1
while (true) {
 nonCriticalSection1;
 c1=0;
 while (c2!=1)
   if (turn==2){
     c1=1;
     while (turn!=1) {}
     c1=0;
   }
 criticalSection1;
 c1=1; turn=2;
}
end P1;
```

```
process P2
while (true) {
 nonCriticalSection2;
 c2=0;
 while (c1!=1)
   if (turn==1){
     c2=1;
     while (turn!=2) {}
     c2=0;
   }
 criticalSection2;
 c2=1; turn=1;
}
end P2;
```

The threads now take turns at backing off.

Dekker’s Algorithm

Mutual exclusion	No deadlock	No starvation	No starvation in absence of contention
✓	✓	✓	✓

Exercise

Consider the following proposed solution to the mutual exclusion problem.

```
int p=1, q=1;
```

```
   process P {
      while (true) {
a1:      non-critical-1;
b1:      while (q!=1) {}
c1:      p=0;
d1:      critical-1;
e1:      p=1;
   }
```

```
   process Q {
      while (true) {
a2:      non-critical-2;
b2:      while (p!=1) {}
c2:      q=0;
d2:      critical-2;
e2;      q=1;
   }
```

Exercise

For each of (i) and (ii) below justify your answer

(i) Does the algorithm operate correctly in the absence of contention?

(ii) Can the algorithm deadlock?

Peterson's algorithm

- Both the turn variable and the status flags are used, as in Dekker's algorithm.
- After setting our flag we immediately give away the turn. We then wait for the turn if and only if the other flag is set.
- By waiting on the **and** of two conditions, we avoid the need to clear and reset the flags.
- Possible for n number of processes

Peterson’s Algorithm

```
int c1=1;
int c2=1;
int turn=1;
```

```
process P1
while (true) {
 nonCriticalSection1;
 c1=0;
 turn=2
       while (turn==2 &&
c2==0) {};

 criticalSection1;
 c1=1;
}
end P1;
```

```
process P2
while (true) {
 nonCriticalSection2;
 c2=0;
 turn=1
       while (turn==1 &&
c1==0) {};

 criticalSection2;
 c2=1;
}
end P2;
```

```
do { flag [i] = TRUE;
       turn = j;
       while ( flag [j] && turn == j); //Note: empty
   loop
              CRITICAL SECTION
       flag [i] = FALSE;
              REMAINDER SECTION
      } while (TRUE);
```

Hardware Assisted Mutual Exclusion

The difficulty in implementing a solution to the mutual exclusion property is caused by possible interleaving of load and store instructions. This difficulty disappears if a load and store are allowed in a single atomic instruction. This is provided on some computers as a test-and-set machine language instruction.

`testAndSet(Li)` is equivalent to the atomic execution of

```
Li=c;
c=1;
```

where `c` is a global variable with initial value `0`; and `Li` is a variable local to process Pi.

Hardware Assisted Mutual Exclusion

```
int c=0;
```

```
process Pi
  private int Li = 0;
  while (true) {
    nonCriticalSectioni;
    do {
      testAndSet(Li);
    } while (Li != 0);
    criticalSectioni;
    c = 0;
  }
end Pi;
```

```
Li=c;
c=1;
```

Test_and_Set solution to the mutual exclusion property for N processes

Hardware Assisted Mutual Exclusion

```
int c=0;
```

```
process Pi
  private int Li = 1;
  while (true) {
    nonCriticalSectioni;
    do {
      exchange(c,Li);
    } while (Li != 0);
    criticalSectioni;
    exchange(c,Li);
  }
end Pi;
```

```
temp=c;
c=Li;
Li=temp;
```

Exchange solution to the mutual exclusion property for N processes

Exercise

Consider the following program that uses the *Exchange* (hardware) instruction described in the notes.

```
int C=0;
```

```
   process P {
      int p=1;
      while (true) {
a1:      non-critical-1;
         while (true) {
b1:         Exchange(C,p);
c1:         if (p==0) {break;}
         }
d1:      critical-1;
e1:      Exchange(C,p);
      }
   }
```

```
   process Q {
      int q=1;
      while (true) {
a2:      non-critical-2;
         while (true) {
b2:         Exchange(C,q);
c2:         if (q==0) {break;}
         }
d2:      critical-2;
e2:      Exchange(C,q);
      }
   }
```

Exercise

(i) Show that the above solution can lead to starvation.

(ii) Show that the program satisfies the mutual exclusion property. [Hint: consider the number of variables having the value 1 at any given time and then consider the implication of both processes being in their critical section at the same time.]

Exercise

Solution

(i) After the statements:

a1; a2; b1; b2;

the value of p is 0 and the values of q and C are both 1. We can then execute the sequence of statements:

c1; d1; e1; a1; b1; c2; b2;

infinitely often which shows that P can enter its critical section infinitely often while Q is being starved.

Exercise

Solution

(ii) There are three variables in the program that all have the value 0 or 1. Initially there are two variables (p and q) which have the value 1 and one which has the value 0 (C). The number of variables that have the value 1 is NOT changed by execution of the *Exchange* instruction (which is the ONLY mechanism by which a variable may have its value changed during program execution). Thus, there are ALWAYS two variables which have the value 1.

Suppose that P and Q are both in their critical sections. Then, from the program code, BOTH p and q must have the value 0, contradicting the observation above.

Thus the mutual exclusion property IS satisfied.

Liveness

- Processes may have to wait indefinitely while trying to acquire a synchronization tool such as a mutex lock or semaphore.
- Waiting indefinitely violates the progress and bounded-waiting criteria discussed at the beginning of this chapter.
- **Liveness** refers to a set of properties that a system must satisfy to ensure processes make progress.
- Indefinite waiting is an example of a liveness failure.

Liveness

- **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

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

- Consider if P_0 executes wait(S) and P_1 wait(Q). When P_0 executes wait(Q), it must wait until P_1 executes signal(Q)
- However, P_1 is waiting until P_0 execute signal(S).
- Since these signal() operations will never be executed, P_0 and P_1 are **deadlocked**.

Liveness

- Other forms of deadlock:
- **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**

End of Chapter 6