Process Synchronization
Critical Section Problem

Multiple Processes

- Central to the design of modern Operating Systems is managing multiple processes
  - Multiprogramming
  - Multiprocessing
  - Distributed Processing
- Big Issue is Concurrency
  - Managing the interaction of all of these processes

Concurrency

- Concurrency arises in:
  - Multiple applications
    - Sharing time
  - Structured applications
    - Extension of modular design
  - Operating system structure
    - OS themselves implemented as a set of processes or threads

Concurrency

- Concurrent access to shared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.

Key Terms

atomic operation	A sequence of one or more statements that appears to be indivisible; that is, no other process can see an intermediate state or interrupt the operation.
critical section	A section of code within a process that requires access to shared resources and that must not be executed while another process is in a corresponding section of code.
deadlock	A situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something.
livelock	A situation in which two or more processes continuously change their states in response to changes in the other process(es) without doing any useful work.
mutual exclusion	The requirement that when one process is in a critical section that accesses shared resources, no other process may be in a critical section that accesses any of those shared resources.
race condition	A situation in which multiple threads or processes read and write a shared data item and the final result depends on the relative timing of their execution.
starvation	A situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen.

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.
- Structure of process P_i

repeat

entry section

critical section

exit section

reminder section

until *false*;

Solution to Critical-Section Problem

- **Mutual Exclusion**. If process *Pi* is executing in its critical section, then no other processes can be executing in their critical sections.
- **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.
- **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.

Initial Attempts to Solve Problem

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

repeat

entry section

critical section

exit section

reminder section

until *false*;

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

Algorithm 1

- Shared variables:
  - **var** *turn*: (0..1);
    initially *turn* = 0
  - *turn* - *i* ⇒ P_i can enter its critical section
- Process P_i

repeat

while *turn* ≠ *i* **do** *no-op*;

critical section

turn := *j*;

reminder section

until *false*;

- Satisfies mutual exclusion, but not progress

Algorithm 2

- Shared variables
  - **var** *flag*: **array** [0..1] **of** *boolean*;
    initially *flag* [0] = *flag* [1] = *false*.
  - *flag* [*i*] = *true* ⇒ P_i ready to enter its critical section
- Process P_i

repeat

flag[*i*] := *true*;
while *flag*[*j*] **do** *no-op*;

critical section

flag [*i*] := *false*;

remainder section

until *false*;

- Satisfies mutual exclusion, but not progress requirement.

Algorithm 3

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

repeat

flag [*i*] := *true*;
turn := *j*;
while (*flag* [*j*] and *turn* = *j*) **do** *no-op*;

critical section

flag [*i*] := *false*;

remainder section

until *false*;

- Meets all three requirements; solves the critical-section problem for two processes.

Race Condition

- A race condition occurs when
  - Multiple processes or threads read and write data items
  - They do so in a way where the final result depends on the order of execution of the processes.
- The output depends on who finishes the race last.
- The race condition are prevented by requiring that critical section be protected by locks.
- That is, a process must acquire a lock before entering a critical section; it releases the lock when it exits the critical section.

Solution to the critical-section problem using locks

do {

acquire lock

critical section

release lock

remainder section

} while (TRUE);

Requirements for Mutual Exclusion

- Only one process at a time is allowed in the critical section for a resource
- A process that halts in its noncritical section must do so without interfering with other processes
- No deadlock or starvation

Semaphore

- The semaphore is used to protect any resource such as global shared memory that needs to be accessed and updated by many processes simultaneously.
- Semaphore acts as ***a guard or lock*** on the resource.
- Whenever a process needs to access the resource, it first needs to take permission from the semaphore.
- The semaphore is implemented as ***an integer variable***, say as ***S***, and can be initialized with any positive integer values.

Semaphore

- The semaphore is accessed by only two indivisible operations known as ***wait()*** and ***signal()*** operations, denoted by P and V, respectively.
- Whenever a process tries to enter the critical section, it needs to perform wait operation.

```
wait(S) {
    while S <= 0
       ; // no-op
    S--;
}
```

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

Semaphore

- Operating systems often distinguish between counting and binary semaphores.
- The value of a **counting semaphore** can range over an unrestricted domain.
- The value of a **binary semaphore** can range only between 0 and 1.
- On some systems, binary semaphores are known as ***mutex locks***, as they are locks that provide mutual exclusion.

Semaphore

- We can use binary semaphores to deal with the critical-section problem for multiple processes.
- Then processes share a semaphore, mutex, initialized to 1.
- Each process is organized as below.

```
do {
   wait(mutex);

      // critical section

   signal(mutex);

      // remainder section
} while (TRUE);
```

Semaphore

- Counting semaphores can be used to control access to a given resource consisting of a finite number of instances.
- The semaphore is initialized to the number of resources available.
- Each process that wishes to use a resource performs a wait() operation on the semaphore (thereby decrementing the count).
- When a process releases a resource, it performs a signal() operation (incrementing the count).
- When the count for the semaphore goes to 0, all resources are being used.
- After that, processes that wish to use a resource will block until the count becomes greater than 0.

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.