Concurrency: Mutual Exclusion and Synchronization
Concurrency

- Communication among processes
- Sharing resources
- Synchronization of multiple processes
- Allocation of processor time
Difficulties with Concurrency

- Sharing global resources
- Management of allocation of resources
- Programming errors difficult to locate
Example: Race Conditions

- Two processes want to access shared memory at the same time
- Solution: Synchronize Access
Operating System Concerns

• Result of processing must be independent of the speed of execution

• Process Interactions:
  - Processes unaware of each other
    • Compete for resources
  - Processes indirectly aware of each other
    • Cooperation by sharing
  - Process directly aware of each other
    • Cooperation by communication
Competition for Resources

• Problem: *Execution of one process may affect the behavior of competing processes*

• Solution: *Mutual Exclusion*
  - a *critical section* is the fragment of code that accesses shared data
  - when one process is executing in its critical section, no other process is allowed to execute in its critical section

• Issue: *Deadlock and Starvation*
  - *blocked process will never get access to the resource and never terminate*
Mutual Exclusion Requirements

• **Mutually exclusive access to critical section**

• **Progress.** If no process is executing in its critical section and there exist some processes that wishes 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 each process executes at a nonzero speed
  - No assumption concerning relative speed of \( n \) processes.
Cooperation by Sharing

- Processes use and update shared data such as shared variables, files, and databases.
- Writing must be *mutually exclusive*.
- *Critical sections* are used to provide data integrity.
Cooperation by Communication

• Communication provides a way to *synchronize*, or coordinate, the various activities
  - no sharing => mutual exclusion not required

• Possible to have *deadlock*
  - each process waiting for a message from the other process

• Possible to have *starvation*
  - two processes sending message to each other while another process waits for a message
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**.
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`
  - `turn - i` ⇒ \( P_i \) can enter its critical section
- Process \( P_i \)
  ```
  do {
      while (turn != i) ;
      critical section
      turn = j;
      reminder section
  } while (1);
  ```
- Satisfies mutual exclusion, but not progress
Algorithm 2

- Shared variables
  - flag [i] = true ⇒ $P_i$ ready to enter its critical section

- Process $P_i$
  do {
    flag[i] := true; 
    while (flag[j]) ;
    critical section
    flag[i] = false;
    remainder section
  } while (1);

- Satisfies mutual exclusion, but not progress requirement.
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.
Mutual Exclusion - Interrupt Disabling

- Process runs until requests OS service or interrupted
- Process disables interrupts for MUTEX
- Processor has limited ability to interleave programs
- Efficiency of execution may be degraded
- Multiprocessing
  - disabling interrupts on one processor will not guarantee mutual exclusion
Mutual Exclusion Instructions

• **Special Machine Instructions**
  - Performed in a single instruction cycle
  - Not subject to interference from other instructions
  - Reading and writing
  - Reading and testing
Test and Set Operation

(Return original value of lock)

boolean TestAndSet (boolean &lock) {
    boolean tmp = lock;
    lock = True;
    return tmp;
}

• When calling TestAndSet(lock)
  - if lock == False before calling TestAndSet
    • it is set to True and False is returned
  - if lock == True before calling TestAndSet
    • it is set to True and True is returned
Mutual Exclusion with Test-and-Set

- **Shared data:**
  
  ```
  boolean lock = False;
  ```

- **Process** $P_i$

  ```
  do {
    while (TestAndSet(lock) == True)
      continue; // do nothing

    -- critical section --
    lock = False;

    -- remainder section
  } while (1);
  ```
Machine Instructions

• Advantages
  - Applicable to any number of processes on either a single processor or multiple processors sharing main memory
  - It is simple and therefore easy to verify
  - It can be used to support multiple critical sections
Mutual Exclusion Instructions

• Disadvantages
  - Busy-waiting consumes processor time
  - Starvation is possible when a process leaves a critical section and more than one process is waiting. *Who is next?*
  - *Deadlock* - If a low priority process has the critical region and a higher priority process needs, the higher priority process will obtain the processor to wait for the critical region
Intro to Semaphores

- Synchronization tool that does not require busy waiting
- Integer variable accessable via two indivisible (atomic) operations:
  - \textit{wait}(s): } s = s - 1, \text{ if } s < 0 \text{ then block thread on the semaphore queue (wait)}
  - \textit{signal}(s): } s = s + 1, \text{ if } s \leq 0 \text{ then wake one sleeping process (signal)}

- \textit{Each Semaphore has an associated queue}.
- Can define a non-blocking version of \textit{wait}(s).
Critical Section of \( n \) Processes

- Shared data:
  
  `semaphore mutex; // initially mutex = 1`

- Process \( P_i \):

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

• Define a semaphore as a record
  
  ```
  typedef struct {
      int value;
      struct process *L;
  } semaphore;
  ```

• Assume two simple operations:
  - `block` suspends the process that invokes it.
  - `wakeup(P)` resumes the execution of a blocked process `P`. 
Implementation

wait(S):
    S.value--;
    if (S.value < 0) {
        add this process to S.L;
        block;
    }

signal(S):
    S.value++;
    if (S.value <= 0) {
        remove a process P from S.L;
        wakeup(P);
    }
Some Reference Background

• Threads are woken up in FIFO order (convoys)

• Used to provide
  - Mutual exclusion (initialized to 1)
  - Event-waiting (initialized to 0)
  - Resource counting (initialized to number available)
Classical Problems of Synchronization

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

• Shared data

    semaphore full, empty, mutex;

Initially:

    full = 0, empty = n, mutex = 1
Bounded-Buffer Problem Producer

do {
    ... produce an item in nextp ... 
    wait(empty);  // Decrement free cnt
    wait(mutex);   // Lock buffers
    ... add nextp to buffer ...
    signal(mutex); // release buffer lock
    signal(full);  // Increment item count
} while (1);
Bounded-Buffer Problem Consumer

do {
    wait(full) // decrement item count
    wait(mutex); // lock buffers
    ... remove item from buffer nextc ...
    signal(mutex); // release lock
    signal(empty); // increment free count
    ... consume item in nextc ...
} while (1);
Readers-Writers Problem

• Shared data

```plaintext
semaphore mutex, wrt;
```

Initially

```plaintext
mutex = 1, wrt = 1, readcount = 0
```
Readers-Writers Problem: Writer

\begin{verbatim}
wait(wrt);
...
writing is performed
...
signal(wrt);
\end{verbatim}
Readers-Writers Problem: Reader

```c
wait(mutex);
readcount++;
if (readcount == 1) // First reader
    wait(wrt); // Keeps writes out
signal(mutex);
... reading is performed ...
wait(mutex);
readcount--; 
if (readcount == 0) // Last reader
    signal(wrt); // Allow writer in
signal(mutex);
```
RW Locks

• Preferred solution, writer wakes up all sleeping readers.
  - But, this could lead to writer starvation. How is this fixed?

• What about when a reader wants to upgrade to an exclusive lock? Deadlocks?

• If pending writers, should a new reader get a lock?
Dining-Philosophers Problem

- Shared data
  
  ```c
  semaphore chopstick[5];
  Initially all values are 1
  ```
Dining-Philosophers Problem

- Philosopher $i$:
  
  ```c
  do {
    wait(chopstick[i])
    wait(chopstick[(i+1) % 5])
    ... eat ...
    signal(chopstick[i]);
    signal(chopstick[(i+1) % 5]);
    ... think ...
  } while (1);
  ```
Potential Problems

• Incorrect use of semaphores can lead to problems
  - Critical section using semaphores: must keep to a strict protocol
  - wait(S); {critical section}; signal(S)

• Problems:
  - No mutual exclusion:
    • Reverse: signal(S); {critical section}; wait(S);
    • Omit wait(S)
  - Deadlock:
    • wait(S); {critical section}; wait(S);
    • Omit signal(S)
Potential Solutions

• How do we protect ourselves from these kinds of errors?
• Develop language constructs that can be validated automatically by the compiler or run-time environment
  - Critical Regions
  - Monitors
Critical Regions

• High-level synchronization construct

• A shared variable $v$ of type $T$, is declared as:

  $v: \text{shared } T$

• Variable $v$ accessed only inside statement region $v$ when $B$ do $S$

  where $B$ is a Boolean expression.

• While statement $S$ is being executed, no other process can access variable $v$. 
Critical Regions

• Regions referring to the same shared variable exclude each other in time.

• When a process tries to execute the region statement, the Boolean expression $B$ is evaluated. If $B$ is true, statement $S$ is executed. If $B$ is false, the process is delayed until $B$ becomes true and no other process is in the region associated with $v$. 
Example – Bounded Buffer

• Shared data:

```c
struct buffer {
    int pool[n];
    int count, in, out;
}
```
Bounded Buffer Producer

- Producer process inserts `nextp` into the shared buffer

```plaintext
region buffer when (count < n) {
    pool[in] = nextp;
    in:= (in+1) % n;
    count++;
}
```
Bounded Buffer Consumer

- Consumer process removes an item from the shared buffer and puts it in `nextc`

```c
region buffer when (count > 0) {
    nextc = pool[out];
    out = (out+1) % n;
    count--;
}
```
Dining Philosophers Example

```c
monitor dp {
    enum {thinking, hungry, eating} state[5];
    condition self[5];
    void pickup(int i) // following slides
    void putdown(int i) // following slides
    void test(int i) // following slides
    void init() {
        for (int i = 0; i < 5; i++)
            state[i] = thinking;
    }
}
```
Dining Philosophers

void pickup(int i) {
    state[i] = hungry;
    test(i);
    if (state[i] != eating)
        self[i].wait();
}

void putdown(int i) {
    state[i] = thinking;
    // test left and right neighbors
    test((i+4) % 5);
    test((i+1) % 5);
}
void test(int i) {
    if ( (state[(i + 4) % 5] != eating) &&
        (state[i] == hungry) &&
        (state[(i + 1) % 5] != eating)) {
        state[i] = eating;
        self[i].signal();
    }
}