Cooperating Processes

- *Independent* process cannot affect or be affected by the execution of another process.

- *Cooperating* process can affect or be affected by the execution of another process.

- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience
Producer-Consumer Problem

- Share the variables
- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process.
  - *bounded-buffer* (circular array) assumes that there is a fixed buffer size.
  - A variable *counter*, initialized to 0 and incremented each time a new item is added to the buffer
Problem

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

```c
item nextConsumed;

while (1) {
    while (counter == 0)
        ; /* do nothing */
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
}
```
The following statements must be performed *atomically*.

```c
counter++;
register1 = counter
register1 = register1 + 1
counter = register1

counter--;
register2 = counter
register2 = register2 - 1
counter = register2
```
Bounded Buffer

- If both the producer and consumer attempt to update the buffer concurrently, the assembly language statements may get interleaved.

- Interleaving depends upon how the producer and consumer processes are scheduled.
The Critical-Section Problem

- **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**.
- Each process has a code segment, called **critical section**, in which the shared data is accessed.
- **Mutual Exclusion** – ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.
Semaphores

- Semaphore S – integer variable
- can only be accessed via two indivisible (atomic) operations. (S initialized to be the number of concurrent processes allowed. S==1 ⇒ Mutex)

**wait (S):**

```plaintext
while S≤ 0 do no-op;
S--;
```

**signal (S):**

```plaintext
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);
  ```
Semaphore Implementation: SpinLock

- Busy waiting
  - Waste of CPU
- Useful with Multiple Processors and short lock time
  - Context Switch is expensive
- Disable interrupt and use atomic operations with SMP

```c
spin_lock:
1: lock; decb slp
   jns 3f
2: cmpb $0 , slp
   pause
   jle 2b
   jmp 1b
3: ...
```

```c
spin_unlock:
Lock; movb $1, slp
```
Semaphore Implementation

- Define a semaphore as a record:
  
  ```c
  typedef struct {
    int value;
    struct process *L; // a queue of PCB
  } semaphore;
  ```

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

- Semaphore operations now defined as
  
  \[ \text{\textit{wait}(S)}: \]
  
  \[
  S.\text{value}--;
  \]
  
  \[
  \text{if } (S.\text{value} < 0) \{ \]
  
  \[
  \text{add this process to } S.L; \]
  
  \[
  \text{block}; \]
  
  \[
  \}
  \]

  \[ \text{\textit{signal}(S)}: \]
  
  \[
  S.\text{value}++; \]
  
  \[
  \text{if } (S.\text{value} <= 0) \{ \]
  
  \[
  \text{remove a process } P \text{ from } S.L; \]
  
  \[
  \text{wakeup}(P); \]
  
  \[
  \}
  \]

- \( S<0 \): its magnitude is the number of waiting processes
Shared data

- `mutex`: mutual exclusion for the critical section
- `full`: the number of full buffers; for synchronization
- `empty`: the number of empty buffers; for synchronization.

```
Semaphore full, empty, mutex;
Initially:
full = 0, empty = n, mutex = 1
```
Bounded-Buffer Problem Producer Process

do {
    ...
    produce an item in nextp
    ...
    wait(empty);
    wait(mutex);
    ...
    add nextp to buffer
    ...
    signal(mutex);
    signal(full);
} while (1);
Bounded-Buffer Problem Consumer Process

do {
    wait(full)
    wait(mutex);
    ...
    remove an item from buffer to nextc
    ...
    signal(mutex);
    signal(empty);
    ...
    consume the item in nextc
    ...
} while (1);
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
Solaris 2 Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing.

- Uses adaptive mutexes for efficiency when protecting data from short code segments.
  - On a multiple processor system, an adaptive mutex starts as a spinlock. If the thread holding the lock is not currently running, the calling thread blocks and sleeps until the lock is released.
  - On a uniprocessor system, the thread always sleep rather than spin.

- Uses condition variables and readers-writers locks when longer sections of code need access to data.
  - Multiple threads may read data concurrently.