### Process Synchronization

- Suppose the value 1000 is stored in ADDRESS1 in main memory.
- Suppose we try to run these processes in parallel:
  
  **Process A**  
  ```
  load a, (ADDRESS1)
  a-=100
  store a, (ADDRESS1)
  ```

  **Process B**  
  ```
  load a, (ADDRESS1)
  a-=100
  store a, (ADDRESS1)
  ```

- If we start with process A and we suspend process A before the store command, our result will be 900 instead of 800.

### Background

- Concurrent access to shared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.
- Shared-memory solution to bounded-buffer problem (Chapter 4) allows at most $n - 1$ items in buffer at the same time. A solution, where all $N$ buffers are used is not simple.
  - 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  
  ```
  type item = ... ;
  var buffer array [0..n-1] of item;
  in, out: 0..n-1;
  counter: 0..n;
  in, out, counter := 0;
  ```

- Producer process  
  ```
  repeat
  ...
  produce an item in nextp
  ...
  while counter = n do no-op;
  buffer [in] := nextp;
  in := in + 1 mod n;
  counter := counter + 1;
  until false;
  ```

### Bounded-Buffer (Cont.)

- Consumer process  
  ```
  repeat
  while counter = 0 do no-op;
  nextc := buffer [out];
  out := out + 1 mod n;
  counter := counter - 1;
  ...
  consume the item in nextc
  ...
  until false;
  ```

- The statements:
  - `counter := counter + 1;`
  - `counter := counter - 1;`
  must be executed atomically.
- Atomic operation means an operation that completes in its entirety without interruption.
Interleaving of Statements

- The statement "counter:=counter+1" may be implemented in machine language as:
  
  load register1 from counter
  register1 = register1 + 1
  store register1 into counter

- The statement "counter:=counter-1" may be implemented as:
  
  load register2 from counter
  register2 = register2 - 1
  store register2 into counter

- Assume counter is initially 5. One interleaving of statements is:
  
  producer: load register1 from counter (register1 = 5)
  producer: register1 = register1 + 1 (register1 = 6)
  consumer: load register2 from counter (register2 = 5)
  consumer: register2 = register2 - 1 (register2 = 4)
  producer: store register1 into counter (counter = 4)

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

Race Condition

- If two or more processes attempt to update the buffer concurrently, the machine language statements may get interleaved.
- Interleaving depends upon how the processes are scheduled.
- Race condition: The situation where several processes access and manipulate shared resource concurrently. The final result depends upon the processes' order of execution. The results may be different on several executions of the same program.
- 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.
- Structure of process Pi
  
  entry section
  critical section
  exit section

Solution to Critical-Section Problem

1. Mutual Exclusion. If process Pi 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 processes that will enter the critical section next cannot be postponed indefinitely. (No deadlock).
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. (No starvation).
   - No assumption concerning relative speed of the n processes.
Initial Attempts to Solve Problem

- Only 2 processes, \( P_0 \) and \( P_1 \).
- In \( P_0 \to i=0 \) and \( j=1 \). In \( P_1 \to i=1 \) and \( j=0 \).
- General structure of process \( P_i \) (other process \( P_j \))

```
entry section
  critical section
exit section
```

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

Algorithm 1

- Shared variables:
  - \texttt{var turn: (0..1); initially turn = 0}
  - \texttt{turn \( \neq i \Rightarrow P_i \) can enter its critical section}
- Process \( P_i \)

```
entry section \rightarrow \texttt{while turn \( \neq i \) do no-op;}
  critical section
exit section \rightarrow turn := j;
```

- Satisfies mutual exclusion and bound waiting, but not progress.

Algorithm 2

- Shared variables
  - \texttt{var flag: array [0..1] of boolean; initially flag [0] = flag [1] = false.}
  - \texttt{flag [i] = true \( \Rightarrow P_i \) ready to enter its critical section}
- Process \( P_i \)

```
entry section \rightarrow \texttt{flag[i] := true; while flag[i] do no-op;}
  critical section
exit section \rightarrow flag [i] := false;
```

- Satisfies mutual exclusion and bounded waiting, but not progress.

Algorithm 3

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

```
entry section \rightarrow \texttt{flag [i] := true; turn := j; while (flag [j] and turn = j) do no-op;}
  critical section
exit section \rightarrow flag [i] := false;
```

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

Critical section for n processes

- Before entering its critical section, process receives a number. Holder of the smallest number enters the critical section.
- If processes $P_i$ and $P_j$ receive the same number, if $i < j$, then $P_i$ is served first; else $P_j$ is served first.
- The numbering scheme always generates numbers in non-decreasing order of enumeration; i.e., 1, 2, 3, 3, 3, 3, 4, 5, ...

Bakery Algorithm (Cont.)

- Notation $\rightarrow$ lexicographical order (ticket #, process id #)
  - $(a, b) < (c, d)$ if $a < c$ or $a = c$ and $b < d$
- Shared data
  - $\text{var choosing: array}[0..n-1]$ of boolean;
  - $\text{number: array}[0..n-1]$ of integer,
Data structures are initialized to false and 0 respectively.

Bakery Algorithm (Cont.)

.entry section

$\text{choosing[i]} := \text{true};$
$\text{number[i]} := \text{max(number[0], number[1], ..., number[n-1])+1;}$
$\text{choosing[i]} := \text{false};$
$\text{for } j := 0 \text{ to } n-1$
$\text{do begin}$
  $\text{while } \text{choosing[j]} \text{ do no-op;}$
  $\text{while } \text{number[i]} \neq 0$
    $\text{and (number[j],j) < (number[i], i) do no-op;}$
.end;

.exit section

$\text{number[i]} := 0;$

Synchronization Hardware

- Test and modify the content of a word **atomically**.

.function $\text{Test-and-Set (var target: boolean): boolean;}$
  \text{begin}$
  $\text{Test-and-Set := target; target := true;}$
  \text{end;}$
Mutual Exclusion with Test-and-Set

- Shared data: `var lock: boolean (initially false)`
- Process $P_i$

\[
\begin{align*}
\text{entry section} & \rightarrow \text{while Test-and-Set (lock) do no-op;} \\
\text{critical section} & \\
\text{exit section} & \rightarrow \text{lock := false;}
\end{align*}
\]

- A possibility of starvation.
- There is a problem of busy waiting.

Mutual Exclusion with Swap

- Shared data (initialized to `false`):
  `var lock: boolean;`
- Unshared data
  `var key: boolean;`
- Process $P_i$

\[
\begin{align*}
\text{entry section} & \rightarrow \text{key := true;} \\
\text{critical section} & \rightarrow \text{while key = true do swap(lock,key);} \\
\text{exit section} & \rightarrow \text{lock = false;}
\end{align*}
\]

Semaphore

- Synchronization tool that implemented in many operation systems and does not require busy waiting.
- Shared variables
  - `var mutex : semaphore`
  - `initially mutex = 1`
- Process $P_i$

\[
\begin{align*}
\text{entry section} & \rightarrow \text{wait(mutex);} \\
\text{critical section} & \\
\text{exit section} & \rightarrow \text{signal(mutex);} \\
\end{align*}
\]

Another Synchronization Hardware

- **Atomically** swap two variables.

\[
\begin{align*}
\text{procedure swap (var a, b: boolean);} & \\
\text{begin} & \\
\text{var temp: boolean;}& \\
\text{temp := a;}& \\
\text{a := } b & \\
\text{b := temp;} & \\
\text{end;}& \\
\end{align*}
\]
Semaphore Implementation

- Define a semaphore as a record
  ```
  type semaphore = record
    value: integer
    L: queue of process;
  end;
  ```
- Assume two simple operations:
  - block suspends the process that invokes it.
  - wakeup(P) resumes the execution of a blocked process P.
- Can only be accessed via two indivisible atomic operations:
  - wait
  - signal

Semaphore Implementation (Cont.)

- Semaphore operations now defined as
  ```
  wait(S): S.value := S.value – 1;
  if S.value < 0 then begin
    add this process to S.L;
    block;
  end;
  ```
  ```
  signal(S): S.value := S.value + 1;
  if S.value ≤ 0 then begin
    remove a process P from S.L;
    wakeup(P);
  end;
  ```

Semaphore as General Synchronization Tool

- Execute B in Pj only after A executed in Pi
- Use semaphore flag initialized to 0
- Code:
  ```
  P_i
  ...
  A
  wait(flag)
  signal(flag)
  B
  ```

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_f
  wait(S);
  wait(Q);
  wait(Q);
  wait(S);
  ...
  signal(S);
  signal(Q);
  signal(Q);
  ```
- Starvation – Indefinite blocking. There is no way to know how much processes will get the CPU before a starved process.
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.
- Can implement a counting semaphore $S$ as a binary semaphore.

Implementing $S$ as a Binary Semaphore

- Data structures:
  
  ```
  var S1: binary-semaphore;
  S2: binary-semaphore;
  C: integer;
  ```

- Initialization:
  
  $$
  S1 = 1
  S2 = 0
  C = \text{initial value of semaphore } S
  $$

Implementing $S$ (Cont.)

- **wait operation:**
  
  ```
  wait(S1);
  C := C - 1;
  if C < 0
  then begin
    signal(S1);
    wait(S2);
  end
  else signal(S1);
  ```

- **signal operation:**
  
  ```
  wait(S1);
  C := C + 1;
  if C < 0 then signal(S2);
  signal(S1);
  ```

Bounded-Buffer Problem

- **Shared data**
  
  ```
  type item = ...;
  var buffer = ...;
  full, empty, mutex: semaphore;
  nextp, nextc: item;
  full := 0; empty := n; mutex := 1;
  ```
Bounded-Buffer Problem (Cont.)

- Producer process

```
repeat
    ...
    produce an item in nextp
    ...
    wait(empty);
    wait(mutex);
    ...
    signal(mutex);
    signal(full);
until false;
```

- Consumer process

```
repeat
    wait(full)
    wait(mutex);
    ...
    remove an item from buffer to nextc
    ...
    signal(mutex);
    signal(empty);
    ...
    consume the item in nextc
    ...
until false;
```

Readers-Writers Problem

- Shared data

```
var mutex, wrt: semaphore (=1);
readcount: integer (=0);
```

- Writer process

```
wait(wrt);
...
writing is performed
...
signal(wrt);
```

- Reader process

```
wait(mutex);
readcount := readcount +1;
if readcount = 1 then wait(wrt);
signal(mutex);
...
reading is performed
...
wait(mutex);
readcount := readcount – 1;
if readcount = 0 then signal(wrt);
signal(mutex):
```

- Possibility of starvation to the writer processes.
Dining-Philosophers Problem

- Shared data
  
  \begin{verbatim}
  var chopstick: array [0..4] of semaphore;
  (=1 initially)
  \end{verbatim}

Dining-Philosophers Problem (Cont.)

- Philosopher \( i \):
  \begin{verbatim}
  repeat
    wait(chopstick[i])
    wait(chopstick[(i+1 mod 5)])
    ... 
    eat
    ... 
    signal(chopstick[i]);
    signal(chopstick[(i+1 mod 5)]);
    ... 
    think
    ... 
  until false;
  \end{verbatim}

- Possibility of deadlock.

Critical Regions

- High-level synchronization construct
- A shared variable \( v \) of type \( T \), is declared as:
  \begin{verbatim}
  var \( v \): shared \( T \)
  \end{verbatim}
- Variable \( v \) accessed only inside statement
  \begin{verbatim}
  region \( v \) when \( B \) do \( S \)
  \end{verbatim}
  
  where \( B \) is a Boolean expression.
  While statement \( S \) is being executed, no other process can access variable \( v \).

Critical Regions (Cont.)

- 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 it is false, the process is delayed until \( B \) becomes true and no other process is in the region associated with \( v \).
- Critical Regions are not the only high level interface. There are some other high level interfaces sometimes used e.g. Monitors.
Example – Bounded Buffer

- Shared variables:

  ```pascal
  var buffer: shared record
  pool: array [0..n–1] of item;
  count,in,out: integer
  end;
  ```

- Producer process inserts `nextp` into the shared buffer

  ```pascal
  region buffer when count < n
  do begin
    pool[in] := nextp;
    in := in + 1 mod n;
    count := count + 1;
  end;
  ```

Bounded Buffer Example (Cont.)

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

  ```pascal
  region buffer when count > 0
  do begin
    nextc := pool[out];
    out := out + 1 mod n;
    count := count – 1;
  end;
  ```