Module 6: Process Synchronization

- Background
- The Critical-Section Problem
- Synchronization Hardware
- Semaphores
- Classical Problems of Synchronization
- Critical Regions
- Monitors
- Synchronization in Solaris 2
- Atomic Transactions

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-butter 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 \( \text{counter} \), 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:
  - \( \text{counter} := \text{counter} + 1; \)
  - \( \text{counter} := \text{counter} - 1; \)
  - must be executed atomically.
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

1. **Mutual Exclusion.** If process $P_i$ 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 processes 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.

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

```
repeat
  while $\text{turn} \neq i$ do
    no-op;
  critical section
  $\text{turn} := j$;
  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 ⇒ Pi ready to enter its critical section

- Process Pi
  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 Pi
  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.

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 Pi and Pj receive the same number, if i < j, then Pi is served first; else Pj is served first.
- The numbering scheme always generates numbers in increasing order of enumeration; i.e., 1,2,3,3,3,3,4,5...

Bakery Algorithm (Cont.)

- Notation <= lexicographical order (ticket #, process id #)
  - (a,b) < (c,d) if a < c or if a = c and b < d
  - max (a0,…, an) is a number, k, such that k ≥ ai for i = 0,…, n − 1

- Shared data
  var choosing: array [0..n – 1] of boolean;
  number: array [0..n – 1] of integer,
  Data structures are initialized to false and 0 respectively
**Bakery Algorithm (Cont.)**

```plaintext
repeat
    choosing[i] := true;
    number[i] := max(number[0], number[1], ..., number[n-1]) + 1;
    choosing[i] := false;
for j := 0 to n - 1
    do begin
        while choosing[j] do no-op;
        while number[j] ≠ 0
            and (number[j], j) < (number[i], i) do no-op;
    end
    critical section
    number[i] := 0;
    remainder section
until false;
```

**Synchronization Hardware**

- Test and modify the content of a word atomically.

```plaintext
function Test-and-Set (var target: boolean): boolean;
begin
    Test-and-Set := target;
    target := true;
end;
```

**Mutual Exclusion with Test-and-Set**

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

```plaintext
repeat
    while Test-and-Set (lock) do no-op;
    critical section
    lock := false;
    remainder section
until false;
```

**Semaphore**

- Synchronization tool that does not require busy waiting.
- Semaphore \( S \) – integer variable
- can only be accessed via two indivisible (atomic) operations

```plaintext
wait (S): while S ≤ 0 do no-op;
S := S - 1;

signal (S): S := S + 1;
```
### Example: Critical Section of n Processes

- **Shared variables**
  - var mutex : semaphore
  - initially mutex = 1
- **Process P_j**
  ```plaintext
  repeat
    wait(mutex);
    critical section
    signal(mutex);
  remainder section
  until false;
  ```

### Semaphore Implementation

- Define a semaphore as a record
  ```plaintext
type semaphore = record
  value: integer
  L: list of process;
end;
  ```
- Assume two simple operations:
  - block suspends the process that invokes it.
  - wakeup(P) resumes the execution of a blocked process P.

### Implementation (Cont.)

- Semaphore operations now defined as
  ```plaintext
  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 P_j only after A executed in P_i
- Use semaphore flag initialized to 0
- Code:
  ```plaintext
  P_i
  ...
  A
  wait(flag)
  ...
  P_j
  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

\[
\begin{align*}
\text{P}_0 & \quad \text{P}_1 \\
\text{wait}(S); & \quad \text{wait}(Q); \\
\text{wait}(Q); & \quad \text{wait}(S); \\
\vdots & \quad \vdots \\
\text{signal}(S); & \quad \text{signal}(Q); \\
\text{signal}(Q) & \quad \text{signal}(S);
\end{align*}
\]

- Starvation – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

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:
  \[\text{var } S1: \text{binary-semaphore}; \]
  \[S2: \text{binary-semaphore}; \]
  \[S3: \text{binary-semaphore}; \]
  \[C: \text{integer}; \]

- Initialization:
  \[S1 = S3 = 1 \]
  \[S2 = 0 \]
  \[C = \text{initial value of semaphore } S \]

Implementing S (Cont.)

- wait operation
  \[
  \begin{align*}
  \text{wait}(S3); \\
  \text{wait}(S1); \\
  C & : = C - 1; \\
  \text{if } C < 0 \text{ then begin} \\
  \quad \text{signal}(S1); \\
  \quad \text{wait}(S2); \\
  \text{end} \\
  \text{else signal}(S1); \\
  \text{signal}(S3);
  \end{align*}
  \]

- signal operation
  \[
  \begin{align*}
  \text{wait}(S1); \\
  C & : = C + 1; \\
  \text{if } C \leq 0 \text{ then signal}(S2); \\
  \text{signal}(S1);
  \end{align*}
  \]
Classical Problems of Synchronization

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

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;
  ```

Bounded-Buffer Problem (Cont.)

• 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
  \[\text{var } \text{mutex}, \text{wrt}: \text{semaphore}(=1);\]
  \[\text{readcount}: \text{integer}(=0);\]

- Reader process
  \[\text{wait}(	ext{mutex});\]
  \[\text{readcount} := \text{readcount} + 1;\]
  \[\text{if } \text{readcount} = 1 \text{ then } \text{wait}(	ext{wrt});\]
  \[\text{signal}(	ext{mutex});\]

- Writer process
  \[\text{wait}(	ext{wrt});\]
  \[\text{writing is performed}\]
  \[\text{signal}(	ext{wrt});\]

Readers-Writers Problem (Cont.)

- Reader process
  \[\text{wait}(	ext{mutex});\]
  \[\text{readcount} := \text{readcount} + 1;\]
  \[\text{if } \text{readcount} = 1 \text{ then } \text{wait}(	ext{wrt});\]
  \[\text{signal}(	ext{mutex});\]

- Reader process
  \[\text{reading is performed}\]
  \[\text{wait}(	ext{mutex});\]
  \[\text{readcount} := \text{readcount} - 1;\]
  \[\text{if } \text{readcount} = 0 \text{ then } \text{signal}(	ext{wrt});\]
  \[\text{signal}(	ext{mutex});\]

Dining-Philosophers Problem

- Shared data
  \[\text{var } \text{chopstick}: \text{array}[0..4] \text{ of semaphore};\]

  (=1 initially)

Dining-Philosophers Problem (Cont.)

- Philosopher \(i\)
  \[\text{repeat}\]
  \[\text{wait}(\text{chopstick}[i]);\]
  \[\text{wait}(\text{chopstick}[i+1 \mod 5]);\]
  \[\text{...}\]
  \[\text{eat}\]
  \[\text{...}\]
  \[\text{signal}(\text{chopstick}[i]);\]
  \[\text{signal}(\text{chopstick}[i+1 \mod 5]);\]
  \[\text{...}\]
  \[\text{think}\]
  \[\text{...}\]
  \[\text{until } \text{false};\]
Critical Regions

- High-level synchronization construct
- A shared variable $v$ of type $T$, is declared as:
  
  ```
  var $v$: 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 (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 $v$.

Example – Bounded Buffer

- Shared variables:
  
  ```
  var buffer: shared record
    pool: array [0..n-1] of item;
    count,in,out: integer
  end;
  ```
- Producer process inserts $\text{nextp}$ into the shared buffer
  
  ```
  region buffer when count < $n$
  do begin
    pool[in] := $\text{nextp}$;
    in := $\text{in} + 1 \mod n$;
    count := count + 1;
  end;
  ```

Bounded Buffer Example (Cont.)

- Consumer process removes an item from the shared buffer and puts it in \text{nextc}
  
  ```
  region buffer when count > 0
  do begin
    $\text{nextc} := \text{pool[out]}$;
    out := $\text{out} + 1 \mod n$;
    count := count - 1;
  end;
  ```
Implementation: region \( x \) when \( B \) do \( S \)

- Associate with the shared variable \( x \), the following variables:
  
  ```
  var mutex, first-delay, second-delay: semaphore;
  first-count, second-count: integer,
  ```

- Mutually exclusive access to the critical section is provided by \( mutex \).

- If a process cannot enter the critical section because the Boolean expression \( B \) is false, it initially waits on the \( first-delay \) semaphore; moved to the \( second-delay \) semaphore before it is allowed to reevaluate \( B \).

---

Implementation (Cont.)

- Keep track of the number of processes waiting on \( first-delay \) and \( second-delay \), with \( first-count \) and \( second-count \) respectively.

- The algorithm assumes a FIFO ordering in the queuing of processes for a semaphore.

- For an arbitrary queuing discipline, a more complicated implementation is required.

---

```plaintext
wait(mutex);
while not B do begin
  first-count := first-count + 1;
  if second-count > 0 then signal(second-delay)
  else signal(mutex);
  wait(first-delay);
  first-count := first-count – 1;
  if first-count > 0 then signal(first-delay)
  else signal(second-delay);
  wait(second-delay);
  second-count := second-count – 1;
end;
S;
if first-count >0 then signal(first-delay);
else if second-count >0 then signal(second-delay);
else signal(mutex);
```
Monitors (Cont.)

- To allow a process to wait within the monitor, a condition variable must be declared, as:
  
  ```
  var x, y: condition
  ```

- Condition variable can only be used with the operations `wait` and `signal`.
  - The operation `x.wait;` means that the process invoking this operation is suspended until another process invokes `x.signal;`
  - The `x.signal` operation resumes exactly one suspended process. If no process is suspended, then the signal operation has no effect.

Schematic view of a monitor

Dining Philosophers Example

```pascal
type dining-philosophers = monitor
var state: array [0..4] of ('thinking', 'hungry', 'eating');
var self: array [0..4] of condition;
procedure entry pickup (i: 0..4);
begin
  state[i] := hungry;
  test (i);
  if state[i] ≠ eating then self[i].wait;
end;

procedure entry putdown (i: 0..4);
begin
  state[i] := thinking;
  test (i+4 mod 5);
  test (i+1 mod 5);
end;
```
### Dining Philosophers (Cont.)

**procedure test(k: 0..4);**

begin

if state[k+4 mod 5] ≠ eating

and state[k] = hungry

and state[k+1 mod 5] ≠ eating

then begin

state[k] := eating;

self[k].signal;

end;

end;

begin

for i := 0 to 4

do state[i] := thinking;

end.

### Monitor Implementation Using Semaphores

- **Variables**

  - `var mutex: semaphore (init = 1)`
  - `next: semaphore (init = 0)`
  - `next-count: integer (init = 0)`

- Each external procedure `F` will be replaced by

  - `wait(mutex);`
  - `...`
  - `...`
  - `if next-count > 0
  then signal(next)
  else signal(mutex);`

- Mutual exclusion within a monitor is ensured.

### Monitor Implementation (Cont.)

- For each condition variable `x`, we have:

  - `var x-sem: semaphore (init = 0)`
  - `x-count: integer (init = 0)`

- The operation `x.wait` can be implemented as:

  - `x-count := x-count + 1;
  if next-count >0
  then signal(next)
  else signal(mutex);
  wait(x-sem);
  x-count := x-count – 1;`

- The operation `x.signal` can be implemented as:

  - `if x-count > 0
  then begin
  next-count := next-count + 1;
  signal(x-sem);
  wait(next);
  next-count := next-count – 1;
  end;`
Monitor Implementation (Cont.)

- Conditional-wait construct: `x.wait(c);
  - `c` – integer expression evaluated when the wait operation is executed.
  - value of `c` (priority number) stored with the name of the process that is suspended.
  - when `x.signal` is executed, process with smallest associated priority number is resumed next.

- Check two conditions to establish correctness of system:
  - User processes must always make their calls on the monitor in a correct sequence.
  - Must ensure that an uncooperative process does not ignore the mutual-exclusion gateway provided by the monitor, and try to access the shared resource directly, without using the access protocols.

Solaris 2 Operating System

- 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.
- Uses condition variables and readers-writers locks when longer sections of code need access to data.