CHAPTER 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-buffer problem (Section 4.4) allows at most $n-1$ items in buffer at the same time.

- Suppose that we modify the producer-consumer code (Section 4.6) by adding a variable *counter*, initialized to 0 and incremented each time a new item is added to the buffer.

  The new scheme is illustrated in the following slide.

- Shared data

```
type item = ... ;
var  buffer: array [0..n–1] of item;
in, out: 0..n–1;
counter: 0..n;
in := 0;
out := 0;
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;
```

- 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*.

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

$\boxed{\textit{entry section}}$

critical section

$\boxed{\textit{exit section}}$

remainder section

until false;

A solution to the critical-section problem must satisfy the following three requirements:

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.

- Assumption that each process is executing at a nonzero speed.
- No assumption concerning *relative* speed of the *n* processes.

Trace of initial attempts to solve the problem.

- Only 2 processes, P_0 and P_1

- General structure of process P_i (other process P_j)

repeat

$\boxed{\textit{entry section}}$

critical section

$\boxed{\textit{exit section}}$

remainder section

until false;

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

Algorithm 1

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

  repeat

  $\boxed{\textbf{while } turn \neq i \textbf{ do } no\text{-}op;}$

  critical section

  $\boxed{turn := j;}$

  remainder 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* $\Rightarrow P_i$ ready to enter its critical section

- Process P_i

repeat

$$\boxed{\begin{array}{l} \mathit{flag}[i] := \mathit{true}; \\ \textbf{while } \mathit{flag}[j] \textbf{ do } \mathit{no\text{-}op}; \end{array}}$$

critical section

$$\boxed{\mathit{flag}[i] := \mathit{false};}$$

remainder section

until *false*;

- Does not satisfy the mutual exclusion requirement.

Algorithm 3

- Combined shared variables of algorithms 1 and 2.

- Process P_i

repeat

$\boxed{\begin{array}{l} \textit{flag}[i] := \textit{true}; \\ \textit{turn} := j; \\ \textbf{while } (\textit{flag}[j] \textbf{ and } \textit{turn}=j) \textbf{ do } \textit{no-op}; \end{array}}$

critical section

$\boxed{\textit{flag}[i] := \textit{false};}$

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 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 increasing order of enumeration.

  Example: 1,2,3,3,3,3,4,5...

- Notation $<$ ≡ lexicographical order (ticket #, process id #)

  - $(a,b) < (c,d)$ if $a < c$ or if $a = c$ and $b < d$

  - $max(a_0, ..., a_{n-1})$ is a number, k, such that $k \geq a_i$ for $i = 0, ..., n-1$

Bakery Algorithm

- Shared data
  - **var** *choosing*: **array** [0..*n*–1] **of** *boolean*;
    number: **array** [0..*n*–1] **of** *integer*;
  - initially *choosing*[*i*] = *false*, for *i* = 0,1,..*n*–1
    number[*i*] = 0, for *i* = 0,1,..*n*–1

```
repeat
    choosing[i] := true;
    number[i] := max(number[0],...,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.

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

- Mutual exclusion algorithm

  - Shared data: **var** *lock*: *boolean* (initially *false*)

  - Process P_i

    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

  wait(S): $S := S - 1;$
  if $S < 0$ **then** *block(S)*

  signal(S): $S := S + 1;$
  if $S \leq 0$ **then** *wakeup(S)*

- *block(S)* – results in suspension of the process invoking it.
- *wakeup(S)* – results in resumption of exactly one process that has invoked *block(S)*.

Example: critical section for *n* processes

- Shared variables
  - **var** *mutex* : *semaphore*
  - initially *mutex* = 1

- Process P_i

repeat

$\boxed{\textit{wait}(\textit{mutex});}$

critical section

$\boxed{\textit{signal}(\textit{mutex});}$

remainder section

until *false*;

Implementation of the *wait* and *signal* operations so that they must execute atomically.

- Uniprocessor environment
  - Inhibits interrupts around the code segment implementing the *wait* and *signal* operations.

- Multiprocessor environment
  - If no special hardware provided, use a correct software solution to the critical-section problem, where the critical sections consist of the *wait* and *signal* operations.
  - Use special hardware if available, i.e., *Test-and-Set*:

Implementation of *wait (S)* operation with the *Test-and-Set* instruction:

- Shared variables
  - **var** *lock* : *boolean*
  - initially *lock* = *false*

- Code for *wait*(S):

```
while Test-and-Set(lock) do no-op;
    S := S - 1;
    if S < 0 then
        begin
            lock := false;
            block(S)
        end
    else lock := false;
```

Race condition exists!

Semaphore can be used as general synchronization tool:

- Execute B in P_j only after A executed in P_i
- Use semaphore *flag* initialized to 0
- Code:

P_i	P_j
.	.
.	.
.	.
A	*wait*(*flag*)
signal(*flag*)	B

- 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.

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.

  - data structures:

    var *S1*: *binary-semaphore*;
    S2: *binary-semaphore*;
    S3: *binary-semaphore*;
    C: *integer*;

  - initialization:

    $S1 = S3 = 1$

    $S2 = 0$

    C = initial value of semaphore S.

- *wait* operation

```
wait(S3);
wait(S1);
C := C - 1;
if C < 0
then begin
        signal(S1);
        signal(S2);
    end
else signal(S1);
signal(S3);
```

- *signal* operation

```
wait(S1);
C := C + 1;
if C ≤ 0 then signal(S2);
signal(S1);
```

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;
```

- Producer process

```
repeat
    ...
   produce an item in nextp
    ...
  wait(empty);
  wait(mutex);
    ...
  add nextp to buffer
    ...
  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);

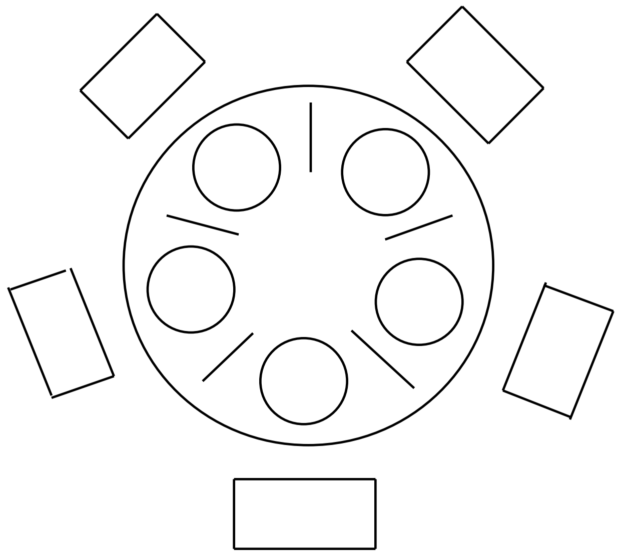
- Writer process

  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*);

Dining-Philosophers Problem



- Shared data

  var *chopstick*: **array** [0..4] **of** *semaphore*;
  (=1 initially)

- Philosopher *i*:

```
repeat
    wait(chopstick[i]);
    wait(chopstick[i+1 mod 5]);
        ...
        eat
        ...
    signal(chopstick[i]);
    signal(chopstick[i+1 mod 5]);
        ...
       think
        ...
until 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*.

- 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 *nextp* into the shared buffer

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

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

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

Implementation of:

region *x* **when** *B* **do** *S*

- We 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*.

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

- The Algorithm

```
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;
         second-count := second-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);
```

- This algorithm assumes a FIFO ordering in the queueing of processes for a semaphore. For an arbitrary queueing discipline, a more complicated implementation is required.

Monitors – high-level synchronization construct that allows the safe sharing of an abstract data type among concurrent processes.

type *monitor-name* = **monitor**
    variable declarations

    procedure entry *P*1 (...);
        begin ... **end**;

    procedure entry *P*2 (...);
        begin ... **end**;

.

.

.

    procedure entry *Pn* (...);
        begin ... **end**;

    begin
        initialization code
    end.

- 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.

```
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;

   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);
    ...
  body of F;
    ...
if next-count > 0
  then signal(next)
  else signal(mutex);
```

- Mutual exclusion within a monitor is ensured.

- 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;

- *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.

- Must check two conditions to establish the correctness of this 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.

Atomic Transactions

- *Transaction* – program unit that must be executed atomically; that is, either all the operations associated with it are executed to completion, or none are performed.

- Must preserve atomicity despite possibility of failure.

- We are concerned here with ensuring transaction atomicity in an environment where failures result in the loss of information on volatile storage.

Log-Based Recovery

- *Write-ahead log* – all updates are recorded on the log, which is kept in stable storage; log has following fields:

  - transaction name
  - data item name, old value, new value

  The log has a record of <T_i **starts**>, and either <T_i **commits**> if the transactions commits, or <T_i **aborts**> if the transaction aborts.

- Recovery algorithm uses two procedures:

  - **undo**(T_i) – restores value of all data updated by transaction T_i to the old values. It is invoked if the log contains record <T_i **starts**>, but not <T_i **commits**>.

  - **redo**(T_i) – sets value of all data updated by transaction T_i to the new values. It is invoked if the log contains both <T_i **starts**> and <T_i **commits**>.

Checkpoints – reduce recovery overhead

1. Output all log records currently residing in volatile storage onto stable storage.

2. Output all modified data residing in volatile storage to stable storage.

3. Output log record <**checkpoint**> onto stable storage.

- Recovery routine examines log to determine the most recent transaction T_i that started executing before the most recent checkpoint took place.
  - Search log backward for first <**checkpoint**> record.
  - Find subsequent <T_i **start**> record.

- **redo** and **undo** operations need to be applied to only transaction T_i and all transactions T_j that started executing after transaction T_i.

Concurrent Atomic Transactions

- *Serial schedule* – the transactions are executed sequentially in some order.

- Example of a serial schedule in which T_0 is followed by T_1:

T_0	T_1
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

- *Conflicting operations* – O_i and O_j *conflict* if they access the same data item, and at least one of these operations is a **write** operation.

- *Conflict serializable* schedule – schedule that can be transformed into a serial schedule by a series of swaps of nonconflicting operations.

- Example of a concurrent serializable schedule:

T_0	T_1
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

- *Locking protocol* governs how locks are acquired and released; data item can be locked in following modes:

  - **Shared:** If T_i has obtained a shared-mode lock on data item Q, then T_i can read this item, but it cannot write Q.

  - **Exclusive:** If T_i has obtained an exclusive-mode lock on data item Q, then T_i can both read and write Q.

- *Two-phase locking protocol*

  - **Growing phase:** A transaction may obtain locks, but may not release any lock.

  - **Shrinking phase:** A transaction may release locks, but may not obtain any new locks.

- The two-phase locking protocol ensures conflict serializability, but does not ensure freedom from deadlock.

- *Timestamp-ordering* scheme – transaction ordering protocol for determining serializability order.
  - With each transaction T_i in the system, associate a unique fixed timestamp, denoted by $\mathrm{TS}(T_i)$.
  - If T_i has been assigned timestamp $\mathrm{TS}(T_i)$, and a new transaction T_j enters the system, then $\mathrm{TS}(T_i) < \mathrm{TS}(T_j)$.
- Implement by assigning two timestamp values to each data item Q.
  - **W-timestamp**(Q) – denotes largest timestamp of any transaction that executed **write**(Q) successfully.
  - **R-timestamp**(Q) – denotes largest timestamp of any transaction that executed **read**(Q) successfully.

- Example of a schedule possible under the timestamp protocol:

T_2	T_3
read(B)	
	read(B)
	write(B)
read(A)	
	read(A)
	write(A)

- There are schedules that are possible under the two-phase locking protocol but are not possible under the timestamp protocol, and vice versa.

- The timestamp-ordering protocol ensures conflict serializability; conflicting operations are processed in timestamp order.