Process Synchronization

- Background
- The Critical-Section Problem
- Synchronization Hardware
- Semaphores
- Classical Problems of Synchronization
- Monitors
- Deadlocks

Background

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

A Simple Example

```
void echo()
{
  chin = getchar();
  chout = chin;
  putchar(chout);
}
```

A Simple Example

```
  Process P1          Process P2

.                       .
in = getchar();         .
.                       in = getchar();
chout = chin;           chout = chin;
putchar(chout);         .
.                       putchar(chout);
.                       .
```

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 *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.
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.

First Attempt

- Busy Waiting
  - Process is always checking to see if it can enter the critical section
  - Process can do nothing productive until it gets permission to enter its critical section

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

  while *turn* $\neq i$ **do** *no-op*;

  critical section

  turn := *j*;

  reminder section

  until *false*;
- Satisfies mutual exclusion, but not progress

Second Attempt

- Each process can examine the other's status but cannot alter it
- When a process wants to enter the critical section is checks the other processes first
- If no other process is in the critical section, it sets its status for the critical section
- This method does not guarantee mutual exclusion
- Each process can check the flags and then proceed to enter the critical section at the same time

Third Attempt

- Set flag to enter critical section before check other processes
- If another process is in the critical section when the flag is set, the process is blocked until the other process releases the critical section
- Deadlock is possible when two process set their flags to enter the critical section. Now each process must wait for the other process to release the critical section

Algorithm 2

- Shared variables
  - **var** *flag*: **array** [0..1] **of** *boolean*;
    initially *flag* [0] = *flag* [1] = *false*.
  - *flag* [*i*] = *true* ⇒ P_i ready to enter its critical section
- Process P_i

  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.

Fourth Attempt

- A process sets its flag to indicate its desire to enter its critical section but is prepared to reset the flag
- Other processes are checked. If they are in the critical region, the flag is reset and later set to indicate desire to enter the critical region. This is repeated until the process can enter the critical region.

Fourth Attempt

- It is possible for each process to set their flag, check other processes, and reset their flags. This scenario will not last very long so it is not deadlock. It is undesirable

Correct Solution

- Each process gets a turn at the critical section
- If a process wants the critical section, it sets its flag and may have to wait for its turn

Algorithm 3

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

  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.

Mutual Exclusion with Test-and-Set

- 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*;

Mutual Exclusion: Hardware Support

- Test and Set Instruction

```
boolean testset (int i) {
if (i == 0) {
i = 1;
return true;
}
else {
return false;
}
}
```

Mutual Exclusion: Hardware Support

- Exchange Instruction

```
void exchange(int register,
                    int memory) {
   int temp;
   temp = memory;
   memory = register;
   register = temp;
}
```

Mutual Exclusion 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 Machine Instructions

- Disadvantages
  - Busy-waiting consumes processor time
  - Starvation is possible when a process leaves a critical section and more than one process is waiting.
  - 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

Semaphores

- Semaphore is a variable that has an integer value
  - May be initialized to a nonnegative number
  - Wait operation decrements the semaphore value
  - Signal operation increments semaphore value

Semaphores

- Special variable called a semaphore is used for signaling
- If a process is waiting for a signal, it is suspended until that signal is sent
- Wait and signal operations cannot be interrupted
- Queue is used to hold processes waiting on the semaphore

Semaphore

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

  wait (*S*): **while** $S \leq 0$ **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_i

```
repeat
        wait(mutex);
                critical section
        signal(mutex);
                remainder section
until false;
```

Semaphore 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 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_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.

Classical Problems of Synchronization

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

Producer/Consumer Problem

- One or more producers are generating data and placing these in a buffer
- A single consumer is taking items out of the buffer one at time
- Only one producer or consumer may access the buffer at any one time

Producer

```
producer:
while (true) {
  /* produce item v */
  b[in] = v;
  in++;
}
```

Consumer

```
consumer:
while (true) {
  while (in <= out)
     /*do  nothing */;
  w = b[out];
  out++;
  /* consume item w */
}
```

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

- Any number of readers may simultaneously read the file
- Only one writer at a time may write to the file
- If a writer is writing to the file, no reader may read it

Readers-Writers Problem

- Shared data

  var *mutex*, *wrt*: *semaphore* (=1);

  readcount : *integer* (=0);

- Writer process

  wait(*wrt*);

  …

  writing is performed

  …

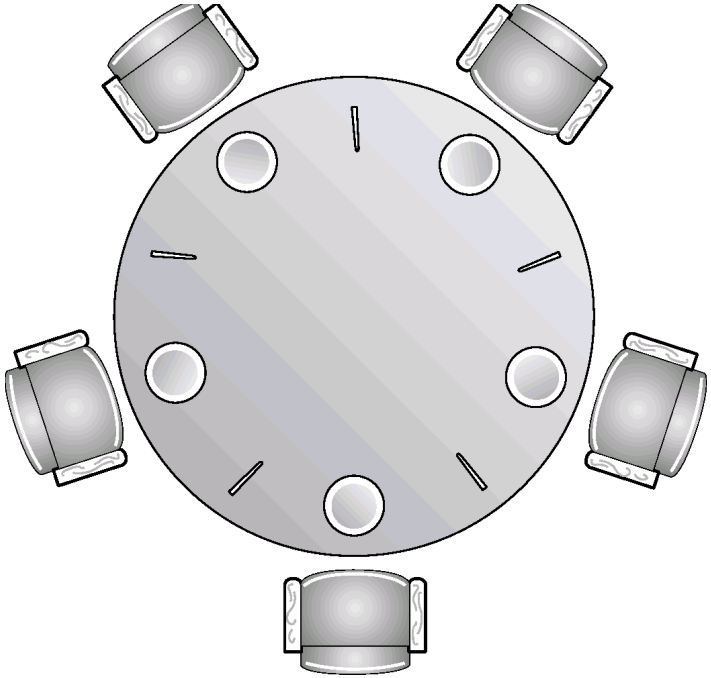
  signal(*wrt*);

Readers-Writers Problem (Cont.)

- 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)

Dining-Philosophers Problem (Cont.)

- 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*;

Monitors

- Monitor is a software module
- Chief characteristics
  - Local data variables are accessible only by the monitor
  - Process enters monitor by invoking one of its procedures
  - Only one process may be executing in the monitor at a time

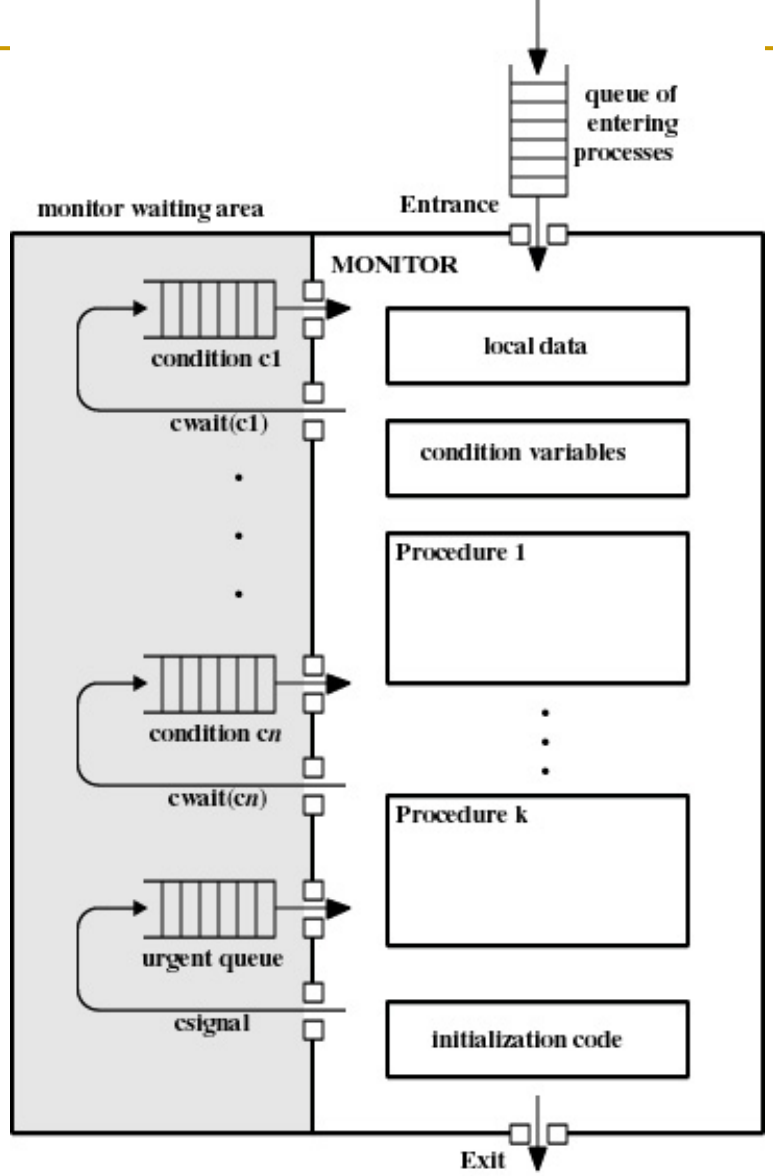


Figure 5.21 Structure of a Monitor

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 *P1* :(…);
begin … **end**;
procedure entry *P2*(…);
begin … **end**;
⋮
procedure entry *Pn* (…);
begin…**end**;
begin
initialization code
end

Monitors (Cont.)

- To allow a process to wait within the monitor, a *condition* variable must be declared, as

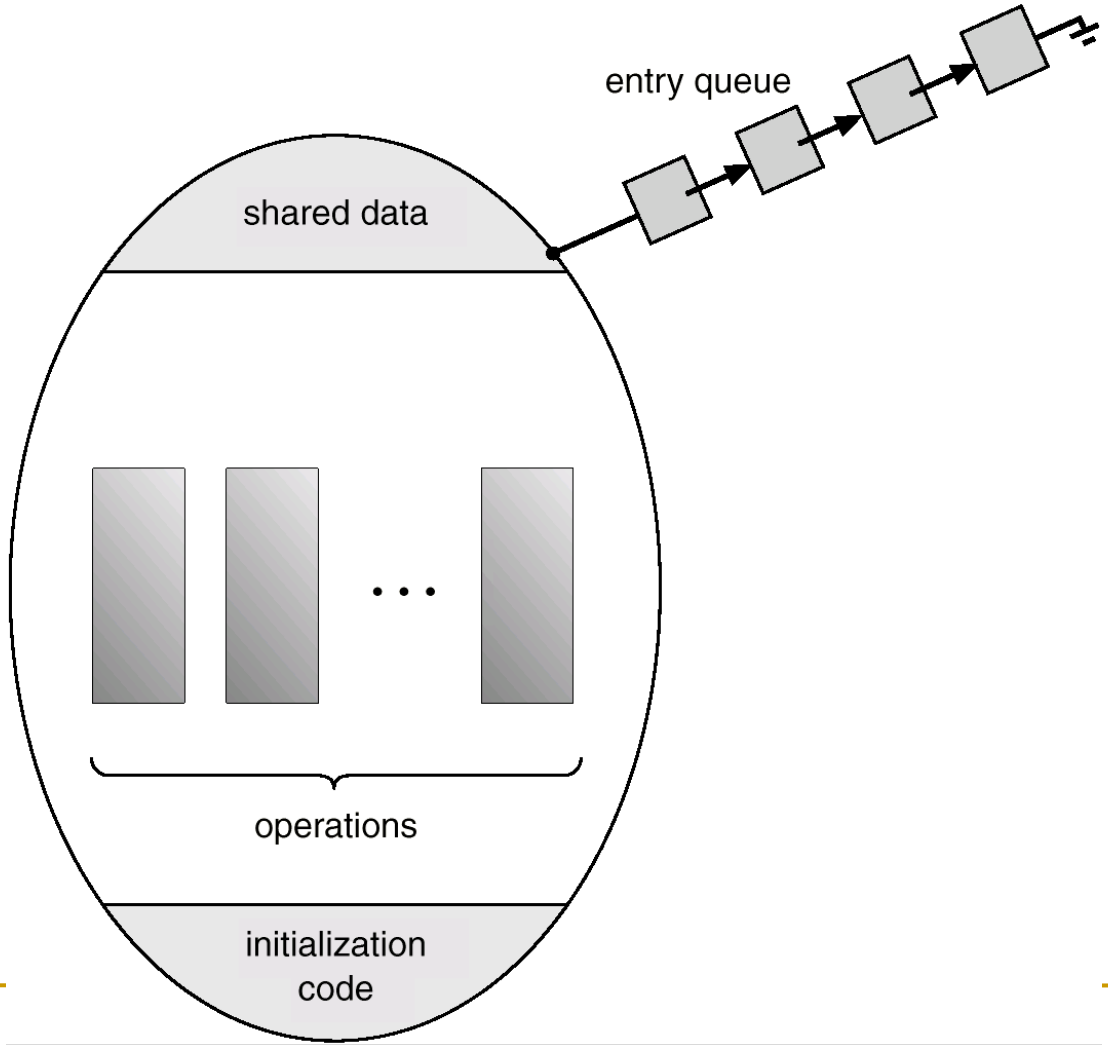
  var *x, y: condition*

- Condition variable can only be used with the operations *wait* and *signal*.
  - The operation

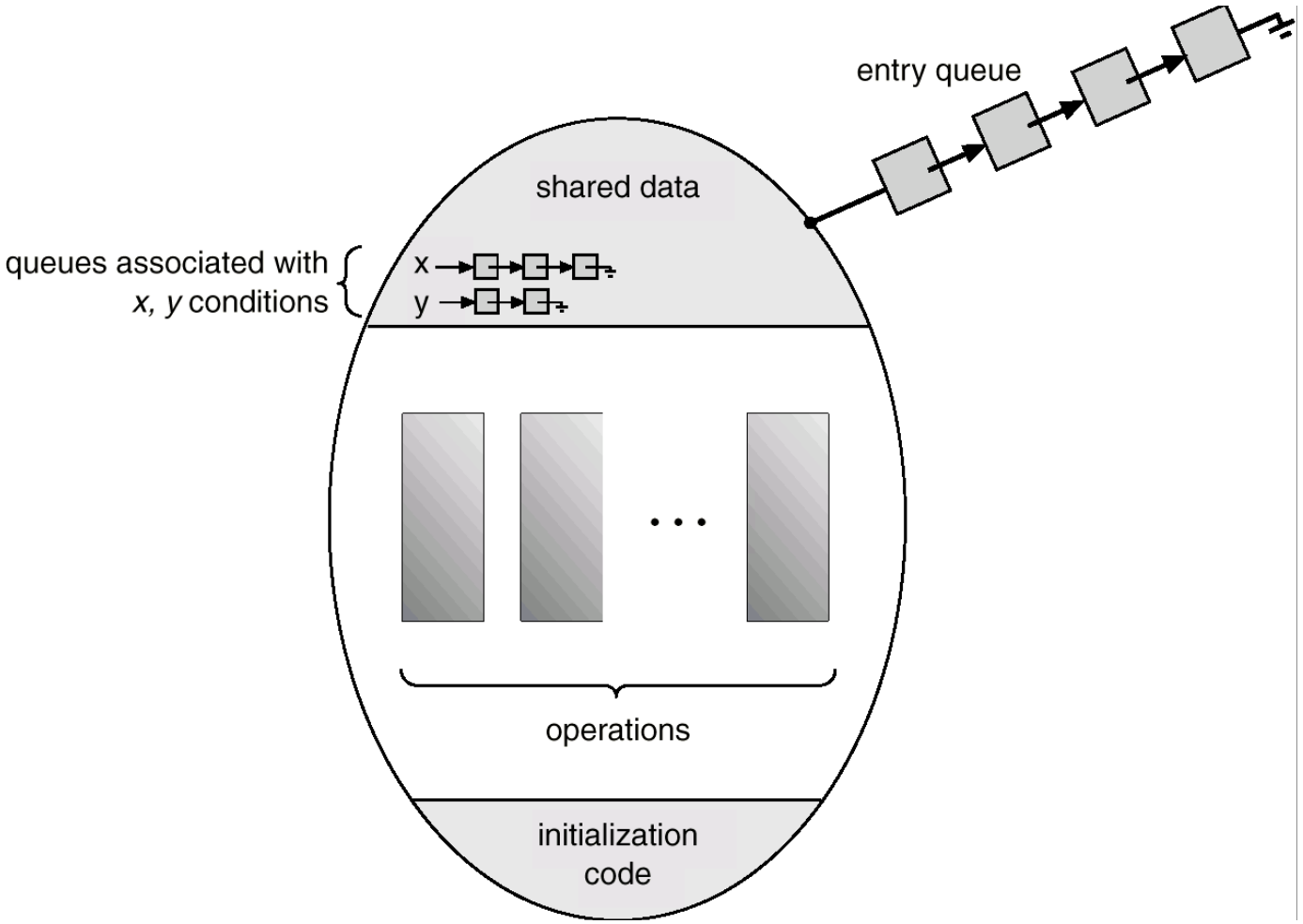
    x.wait;
    means that the process invoking this opeation is suspended until another process invokes

    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



Monitor with condition variables



Dining Philosophers Example

```
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.
```