ICS 143 - Principles of Operating Systems

Lecture 6 and 7 - Process Synchronization

Prof. Nalini Venkatasubramanian
nalini@ics.uci.edu

Outline

- Cooperating Processes
- The Bounded Buffer Producer-Consumer Problem
- The Critical Section Problem
- Synchronization Hardware
- Semaphores
- Classical Problems of Synchronization
- Critical Regions
- Monitors

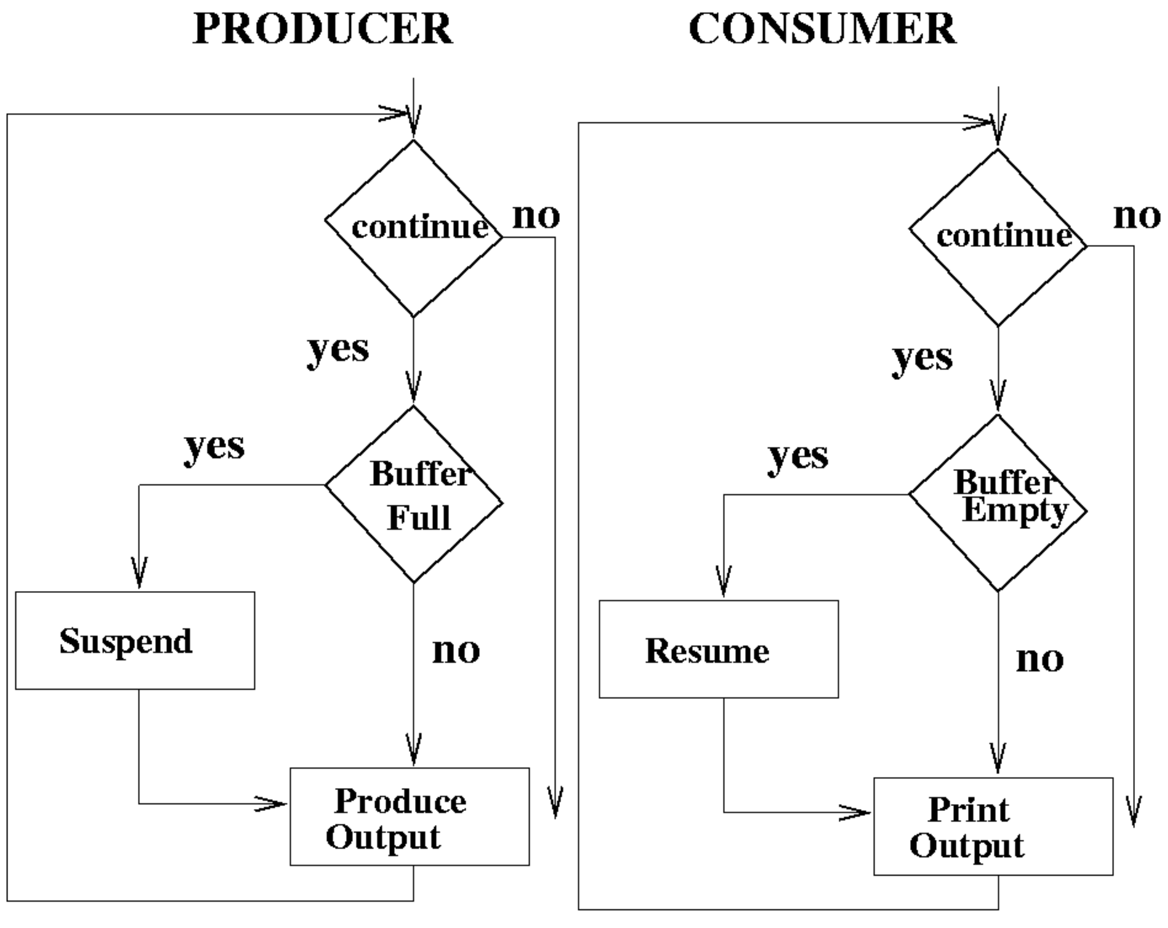
Cooperating Processes

- Concurrent Processes can be
  - Independent processes
    - cannot affect or be affected by the execution of another process.
  - Cooperating processes
    - can affect or be affected by the execution of another process.
- Advantages of process cooperation:
    - Information sharing
    - Computation speedup
    - Modularity
    - Convenience(e.g. editing, printing, compiling)
- Concurrent execution requires
    - process communication and process synchronization

Producer-Consumer Problem

- Paradigm for cooperating processes;
  - producer process produces information that is consumed by a consumer process.
- We need buffer of items that can be filled by producer and emptied by consumer.
    - Unbounded-buffer places no practical limit on the size of the buffer. Consumer may wait, producer never waits.
    - Bounded-buffer assumes that there is a fixed buffer size. Consumer waits for new item, producer waits if buffer is full.
  - Producer and Consumer must synchronize.

Producer-Consumer Problem



Bounded-buffer - Shared Memory Solution

- Shared data

var *n*;

type *item* = ….;

var *buffer*: **array**[0..n-1] **of** *item*;

in, *out*: 0..*n*-1;

in :=0; *out*:= 0; /* shared buffer = circular array */

/* Buffer empty if *in* == *out* */

/* Buffer full if (*in*+1) mod *n* == *out* */

/* *noop* means 'do nothing' */

Bounded Buffer - Shared Memory Solution

- Producer process - creates filled buffers

repeat

…

produce an item in *nextp*

…

while *in*+1 **mod** *n* = *out* **do** *noop*;

buffer[in] := *nextp*;

in := *in+1* **mod** *n*;

until *false*;

Bounded Buffer - Shared Memory Solution

- Consumer process - Empties filled buffers

```
repeat
        while in = out do noop;
    nextc := buffer[out] ;
    out:= out+1 mod n;
  …
  consume the next item in nextc
  …
until false
```

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 the bounded-buffer problem allows at most (n-1) items in the buffer at the same time.

Bounded Buffer

- A solution that uses all N buffers is not that simple.
  - Modify producer-consumer code by adding a variable *counter*, initialized to 0, incremented each time a new item is added to the 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;

Bounded Buffer

- Producer process - creates filled buffers

```
repeat
        …
    produce an item in nextp
      …
    while counter = n do noop;
    buffer[in] := nextp;
    in := in+1 mod n;
    counter := counter+1;
until false;
```

Bounded Buffer

- Consumer process - Empties filled buffers

```
repeat
        while counter = 0 do noop;
    nextc := buffer[out] ;
    out:= out+1 mod n;
    counter := counter - 1;
        …
    consume the next 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 shared data.
  - Structure of process P_i ---- Each process has a code segment, called the critical section, in which the shared data is accessed.

```
repeat
  entry section     /* enter critical section */
      critical section  /* access shared variables */
   exit  section      /* leave critical section */
      remainder section  /* do other work */
until false
```

- Problem
  - Ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.

Solution: Critical Section Problem - Requirements

- **Mutual Exclusion**
  - If process Pi is executing in its critical section, then no other processes can be executing in their critical sections.
- **Progress**
  - If no process is executing in its critical section and there exists 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.
- **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.

Solution: Critical Section Problem - Requirements

- Assume that each process executes at a nonzero speed.
- No assumption concerning relative speed of the n processes.

Solution: Critical Section Problem -- Initial Attempt

- Only 2 processes, P0 and P1
- General structure of process Pi (Pj)

```
repeat
        entry section
                critical section
        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* ☑ P_i can enter its critical section
- Process P_i

```
repeat
    while turn <> i do no-op;
            critical section
    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* ☑ Pi 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
```

Can block indefinitely…. Progress requirement not met.

Algorithm 3

- 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 P_i

```
repeat
    while flag[j] do no-op;
    flag[i] := true;
            critical section
    flag[i]:= false;
            remainder section
until false
```

Does not satisfy mutual exclusion requirement ….

Algorithm 4

- 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
```

YES!!! Meets all three requirements, solves the critical section problem for 2 processes.

Bakery Algorithm

- Critical section for n processes
  - Before entering its critical section, process receives a number. Holder of the smallest number enters critical section.
  - If processes Pi and Pj receive the same number,
    - if i <= j, then P is served first; else Pj is served first.
  - The numbering scheme always generates numbers in increasing order of enumeration; i.e. 1,2,3,3,3,3,4,4,5,5

Bakery Algorithm (cont.)

- Notation -
  - Lexicographic 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 $k >= a_i$ for i = 0,…,n-1
- Shared Data

  var *choosing*: **array**[0..*n*-1] **of** *boolean*;(initialized to *false*)

  number: **array**[0..*n*-1] **of** *integer*; (initialized to 0)

Bakery Algorithm (cont.)

```
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;
```

Hardware Solutions for Synchronization

- Mutual exclusion solutions presented depend on memory hardware having read/write cycle.
    - If multiple reads/writes could occur to the same memory location at the same time, this would not work.
    - Processors with caches but no cache coherency cannot use the solutions
- In general, it is impossible to build mutual exclusion without a primitive that provides some form of mutual exclusion.
    - How can this be done in the hardware???

Synchronization Hardware

- Test and modify the content of a word atomically - Test-and-set instruction

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

- Similarly "SWAP" instruction

Mutual Exclusion with Test-and-Set

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

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

Bounded Waiting Mutual Exclusion with Test-and-Set

```
var j : 0..n-1;
    key : boolean;
repeat
    waiting [i]  := true; key := true;
    while waiting[i] and key do key := Test-and-Set(lock);
    waiting [i ] := false;
 critical section
    j := j + 1 mod n;
    while (j <> i) and (not waiting[j]) do j := j + 1 mod n;
     if j = i  then lock := false;
               else waiting[j] := false;
  remainder section

until false;
```

Semaphore

- Semaphore *S* - integer variable
  - used to represent number of abstract resources
- Can only be accessed via two indivisible (atomic) operations

wait (*S*): **while** *S* <= 0 **do** no-op
S := *S*-1;

signal (S): *S* := *S*+1;

  - *P* or *wait* used to acquire a resource, decrements count
  - *V* or *signal* releases a resource and increments count
  - If *P* is performed on a *count* <= 0, process must wait for *V* or the release of a resource.

Example: Critical Section for 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 a General Synchronization Tool

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

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

Problem...

- Busy Waiting, uses CPU that others could use. This type of semaphore is called a *spinlock*.
  - OK for short times since it prevents a context switch.
- For longer runtimes, need to modify P and V so that processes can *block* and *resume.*

Semaphore Implementation

- Define a semaphore as a record

  type *semaphore* = **record**
  value: *integer*;
  L: **list of** *processes*;
  end;

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

Semaphore Implementation(cont.)

- Semaphore operations are 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;

Block/Resume Semaphore Implementation

- If process is blocked, enqueue PCB of process and call scheduler to run a different process.
- Semaphores are executed atomically;
  - no two processes execute *wait* and *signal* at the same time.
  - Mutex can be used to make sure that two processes do not change count at the same time.
    - If an interrupt occurs while mutex is held, it will result in a long delay.
    - Solution: Turn off interrupts during critical section.

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 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; simpler to implement.
- Can implement a counting semaphore S as a binary semaphore.

Implementing S (counting sem.) 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*;

Implementing S

Wait operation

```
wait(S3);
wait(S1);
C := C-1;
if C < 0
then begin
        signal (S1);
        wait(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*: **array**[0..n-1] **of** *item*;

  full, empty, mutex : *semaphore*;

  nextp, nextc :*item*;

  full := 0; *empty* := *n*; *mutex* := 1;

Bounded Buffer Problem

- Producer process - creates filled buffers

```
repeat
    …
  produce an item in nextp
    …
  wait (empty);
  wait (mutex);
  …
  add nextp to buffer
  …
  signal (mutex);
  signal (full);
until false;
```

Bounded Buffer Problem

- Consumer process - Empties filled buffers

repeat

wait (*full*);

wait (*mutex*);

…

remove an item from *buffer* to *nextc*

...

signal (*mutex*);

signal (*empty*);

…

consume the next item in *nextc*

…

until *false;*

Readers-Writers Problem

- Shared Data

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

  readcount: *integer* (= 0);

- Writer Process

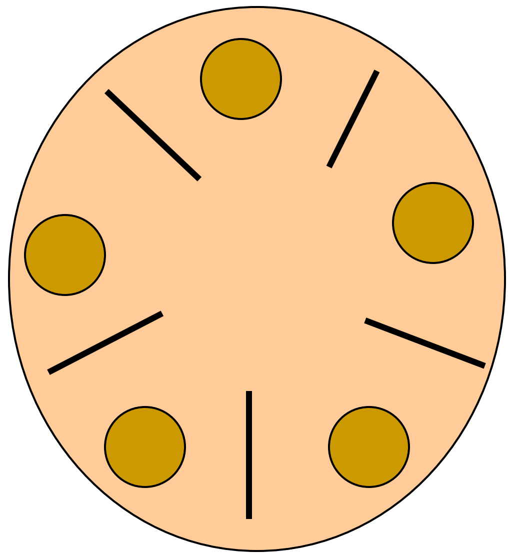
```
wait(wrt);
  …
  writing is performed
  ...
signal(wrt);
```

Readers-Writers Problem

- 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

- 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;
```

Higher Level Synchronization

- Timing errors are still possible with semaphores
  - Example 1

    signal (*mutex*);

    …

    critical region

    ...

    wait (*mutex*);
  - Example 2

    wait(*mutex*);

    …

    critical region

    ...

    wait (*mutex*);
  - Example 3

    wait(*mutex*);

    …

    critical region

    ...

    Forgot to signal

Conditional Critical Regions

- High-level synchronization construct
- A shared variable *v* of type *T* is declared as:

  var *v*: **shared** *T*

- Variable v is accessed only inside statement

  region *v* when *B* **do** <u>*S*</u>

  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 *nextp* into the shared buffer

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

Bounded Buffer Example

- 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;
```

Implementing Regions

- Region *x* when *B* do *S*

  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

- 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 queueing of processes for a semaphore.
- For an arbitrary queueing discipline, a more complicated implementation is required.

Implementing Regions

```
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);
```

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

- 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 within the operations *wait* and *signal*. Queue is associated with condition variable.
    - 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.

Dining Philosophers

```
type dining-philosophers= monitor
    var state: array[0..4] of (thinking, hungry, eating);
    var self:   array[0..4] of condition;
    // condition where philosopher I can delay himself when hungry but     //
     is  unable to obtain chopstick(s)
     procedure entry pickup (i :0..4);
        begin
          state[i] := hungry;
          test(i); //test that your left and right neighbors are not eating
          if state [i] <> eating then self [i].wait;
         end;
     procedure entry putdown (i:0..4);
        begin
              state[i] := thinking;
             test (i + 4 mod 5 );  // signal left neighbor
           test (i + 1 mod 5 );  // signal right  neighbor
        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;
```