Semaphore Use in Synchronization

Ch 6 [BenA 06]

Lesson 7

Semaphore Use In Synchronization

Ch 6 [BenA 06]

Consumer Producer Revisited

Readers and Writers

Baton Passing

Private Semaphores

Resource Management

Semaphore Use in Synchronization

Sem gate = -3; # must know number of R's (!)

Process R[i = 1 to 4]

... V(gate); # signal Q ...

Process Q

... P(gate)

# how to prepare for next time?

# sem_set (gate, -3) ??

sem g[i = 1 to 4] = 0;

Process R[i = 1 to 4]

... V(g[i]); # signal Q ...

Process Q

... P(g[i]); P(g[2]); P(g[3]); P(g[4]); ...

# Q must know number of R's

Barrier Synchronization

with Semaphores

Barrier Synchronization

with Semaphores

Barrier is implemented as separate process

– This is just one possibility to implement the barrier

– Cost of process switches?

– How many process switches?

Barrier Synchronization

with Barrier OS-Primitive

• Specific synchronization primitive in OS

– Implemented with semaphores...

– No need for extra process – less process switches

Barrier Synchronization

with Barrier OS-Primitive

barrier br;

barrier_init (br, 4); # must be done before use

process Q[i = 1 to 4]

... barrier_wait (br) # wait until all have reached this point

if (pid==1) # is this ok? is this done in time?

barrier_init (br, 4) ...

Communication with Semaphores

Communication with Semaphores

Sem mutex=1, data_ready = 0; # one data item buffer

Int buffer

Process W

... P(mutex)

write_buffer(data)

V(mutex)

V(data_ready); # signal Q

... # What is wrong?

W might rewrite data buffer before R reads it

– Might have extra knowledge that avoids the problem

Process R

... P(data_ready); # wait for data

P(mutex)

read_buffer(data)

V(mutex)

V(data_ready); # signal Q ...

Communication with Semaphores Correctly

Communication with Semaphores Correctly

Sem mutex=1, data_ready = 0, buffer_empty=1; # one data item buffer

Int buffer

Process W

... P(mutex)

write_buffer(data)

V(mutex)

V(data_ready); # signal Q

... # Fast W can not overtake R now

– One reader R, one writer W, binary semaphores

– Actual communication with buffer in shared memory

– Use model: 1 producer – 1 consumer – size 1 buffer
Producer-Consumer with Binary Semaphores

• Binary semaphore has values 0 and 1
  – OS or programming language library
• Semaphore does not keep count
  – Must have own variable count (nr of elements in buffer)
  – Protect it with critical section
• Two important state changes
  – Empty buffer becomes not empty
    – Consumer may need to be awakened
  – Full buffer becomes not full
    – Producer may need to be awakened

Simple Solution #1

| typeT buf[n];        /* n element buffer */ |
| int front=0,         /* read from here */  |
|rear=0,              /* write to this one */  |
|count=0;             /* nr of items in buf */  |
|sem space=1,          /* need this to write */  |
|items=0,             /* need this to read */    |
|mutex=1;             /* need this to update count */ |

Sol. #1

process Producer [i=1 to M] {
while(true) {
  ... produce data ...
  P(space); /* wait until space to write*/
  P(mutex);
  buf[rear] = data; rear = (rear+1) %n; count++;
  if (count == 1) V(items); /* first item to empty buffer */
  if (count < n) V(space); /* still room for next producer */
  V(mutex);
}
}

process Consumer [i=1 to N] {
while(true) {
  P(items); /* wait until items to consume */
  P(mutex);
  data=buf[front]; front = (front+1) %n; count--;
  if (count == n-1) V(space); /* buffer was full */
  if (count > 0) V(items); /* still items for next consumer */
  V(mutex);
  ... consume data ...
}
}

Evaluate Solution #1

• Simple solution
  – Mutex and synchronization ok
  – Mutex inside space or items
  • Get space first and then mutex
  • Buffer reserved for one producer/consumer at a time
    – Does not allow for simultaneous buffer use
      • Producer inserts item to “rear”
      • Consumer removes item from “front”
  • First waiting producer/consumer advances when signalled
    – Queued in semaphores

Better Solution #2

| typeT buf[n];        /* n element buffer */ |
| int front=0,         /* read from here */  |
|rear=0,              /* write to this one */  |
|count=0;             /* nr of items in buf */  |
|sem space=1,          /* need this to write */  |
|items=0,             /* need this to read */    |
|mutex=1;             /* need this to update count */ |

Sol. #2

process Producer [i=1 to M] {
while(true) {
  ... produce data ...
  P(space); /* wait until space to write*/
  buf[rear] = data; rear = (rear+1) %n; outside mutex, ok? *
  P(mutex);
  count++;
  if (count == 1) V(items); /* first item to empty buffer */
  if (count < n) V(space); /* still room for next producer */
  V(mutex);
}
}

process Consumer [i=1 to N] {
while(true) {
  P(items); /* wait until items to consume */
  data=buf[front]; front = (front+1) %n; outside mutex, ok? *
  P(mutex);
  count--;
  if (count == n-1) V(space); /* buffer was full */
  if (count > 0) V(items); /* still items for next consumer */
  V(mutex);
  ... consume data ...
}
}
Evaluate Solution #2

- Relatively simple solution
  - Data copying (insert, remove) outside critical section
  - Protected by a semaphore (items and space)
- Simultaneous insert and remove ops
  - Producer inserts item to “rear”
  - Consumer removes item from “front”
- First waiting producer/consumer advances when signalled
  - Queued in semaphores

Another Solution #3

(Producer-Consumer with Binary Semaphores)
- Use condition synchronization
  - Do P(space) or P(items) only when needed
  - Expensive op?
  - Requires execution state change (kernel/user)?

```c
typeT buf[n]; /* n element buffer */
int front=0; /* read from here */
int rear=0; /* write to this one */
int count=0; /* nr of items in buf */
cwp=0; /* nr of waiting producers */
cwc=0; /* nr of waiting consumers */
int space=1; /* need this to write */
int items=0; /* need this to read */
int mutex=1; /* need this to update count */
```

process Producer [i=1 to M] {
    while(true) {
        // produce data …
        P(mutex);
        while (count == n) /* usually not true? while, not if */
            { cwp++; V(mutex); P(space); P(mutex); cwp-- }
        buf[rear] = data; rear = (rear+1) %n;  count++;
        if (count == 1 && cwc>0) V(items);
        if (count < n && cwp>0) V(space);
        V(mutex);
    }
}

process Consumer [i=1 to N] {
    while(true) {
        P(mutex);
        while (count == n) /* while, not if */
            { cwc++; V(mutex); P(items); P(mutex); cwc-- }
        data=buf[front]; front = (front+1) %n; count--;
        if (count == n-1 && cwp>0) V(space);
        if (count > 0 && cwc > 0) V(items);
        V(mutex);
        // consume data …
    }
}
```

Evaluate Solution #3

- No simultaneous insert and remove ops
- Data copying inside critical section
- In general case, only mutex semaphore operations needed
  - Most of the time?
  - Can they be busy-wait semaphores?
- First waiting producer/consumer does not necessarily advance when signalled
  - Someone else may get mutex first
  - E.g., consumer signals V(space), another producer gets (entry) mutex and places its data in buffer.
  - Need “while” loop in waiting code
- Unfair solution even with strong semaphores?
  - How to fix?
  - Baton passing (pass critical section to next process)?
    - Not shown now

Solutions #1, #2, and #3

- Which one is best? Why? When?
- How to maximise concurrency?
  - Separate data transfer (insert, remove) from permission to do it
    - Allow obtaining permission (e.g., code with P(space) and updating count)
      for one process run concurrently with data transfer for another process
        (e.g., code with buf[rear] = data; …)
    - Need new mutexes to protect data transfers and index (rear, front) manipulation
  - Problem: signalling to other producers/consumers should happen in same critical section with updating count, but should happen only after data transfer is completed (i.e., in different critical section …)
Readers and Writers Problem

- Shared data structure or database
- Two types of users: readers and writers
- Readers
  - Many can read at the same time
  - Can not write when someone reads
  - Can not read when someone writes
- Writers
  - Read and modify data
  - Only one can be active at the same time
  - Can be active only when there are no readers

Readers and Writers with Baton

Passing Split Binary Semaphores

- Component semaphores e, r, w
  - Mutex wait in P(e), initially 1
  - Readers wait in P(r) if needed, initially 0
  - Writers wait in P(w) if needed, initially 0
- In critical control areas, only one process advances at a time
  - Wait in e, r, or w
  - One advances, others wait in e, r or w
  - New reader/writer: wait in P(e)
  - Waiting for read turn: V(e); P(r)
    - Wait while not holding mutex
  - Waiting for write turn: V(e); P(w)
    - Wait while not holding mutex
  - When done, pass the baton (turn) to next one

Baton passing

- When done your own mutex zone, wake up next ...
  (one or more semaphores control the same mutex)

  - Do not release mutex (currently reserved e, r, or w)
  - New reader will continue with mutex already locked
  - "Pass the mutex baton to next reader"
    - No one else can come to mutex zone in between
    - Last waiting reader will close the mutex with V(e)
      - Can happen concurrently when reading database
  - Else if writer waiting and no readers: V(w)
    - Do not release mutex, pass baton to writer
  - Else (let new process to compete with old ones): V(e)
    - Release mutex to let new process in the game
      - New process gets in mutex only when no old one can be advance
      - Can happen concurrently when reading database
Concurrent Programming (RIO) 4.2.2011

Lecture 7: Readers and Writers 5

Baton Passing with SIGNAL

```c
if (nw == 0 and dr > 0) {
    dr = dr - 1;
    V(r); // wake up waiting reader
}
else if (nr == 0 and nw == 0 and dw > 0) {
    dw = dw - 1;
    V(w); // wake up waiting writer
}
else
    V(e); // let new process to mix
```

Resource Management

- **Problem**
  - Many types of resources
  - N units of given resource
  - Request allocation: K units
    - Wait suspended until resource available
- **Solution**
  - Semaphore mutex (init 1)
  - Semaphore Xavail
    - init N – wait for available resource
    - init 0 – wait for permission to continue

Simple Very Bad Solution

```c
sem Xmutex = 1, Xavail = N
Xres_request() # one unit at a time
P(Xmutex)
P(Xavail) # ok if always allocate just 1 unit
    # may take long time?
V(Xmutex);
Xres_release()
P(Xmutex)
return 1 unit
V(Xavail);
V(Xmutex);
```

Another Not So Good Solution

```c
sem Xmutex = 1, Xavail = N
Xres_request() # one unit at a time
P(Xmutex) # ok if always allocate just 1 unit
P(Xavail) # ok if always allocate just 1 unit
    # may take long time?
V(Xmutex);
Xres_release()
P(Xmutex)
return 1 unit
V(Xmutex);
V(Xavail);
```

Resource Management with Baton Passing Split Semaphore

```c
sem Xmutex = 1, Xavail = N (not N) ; split semaphore
Xres_request (K) – request K units of given resource
P(Xmutex)
if "not enough free units" {V(Xmutex); P(Xavail);}
take K units ; assume short time
else
    V(Xmutex);
Xres_release (K)
P(Xmutex)
return K units
if "requests pending and enough free units" {V(Xavail);}
else V(Xmutex);
```

Fig. 4.13 [Andr00]: readers / writers solution using passing the baton (with SIGNAL code)
Problems with Resource Management

- Need strong semaphores
- Strong semaphores are FIFO
  - What if 1\textsuperscript{st} in line want 6 units, 2\textsuperscript{nd} wants 3 units, and there are 4 units left?
  - What about priorities?
    - Each priority class has its own semaphore
    - Baton passing within each priority class?
  - How to release just some specific process?
    - Strong semaphore releases 1\textsuperscript{st} in line
    - Answer: private semaphores

Private Semaphore

- Semaphore, to which only one process can ever make a P-operation
  - Initialized to 0, belongs to that process
- Usually part of PCB (process control block) for the process
  - Create own semaphore array for this purpose
- Process makes demands, and then waits in private semaphore for turn
- Most often just one process at a time
  - Usually P(mutex) does not lead to process switches
  - Usually still need to wait in private semaphore

Private Semaphore Use Example

- Common resource allocation method
  - Here: time = amount of resource requested
  - Here: just one job (at most) holding the resource at a time
- Use private semaphores
  - request(time, id): # requested time, user id
  - P(e);
  - if (!free)
    - DELAY(); # wait for your turn
  - free = false;
  - V(e);

Process

- set up resource demands
  - P(mutex)
  - locate next process Q to release
  - V(mutex)

Private Semaphore

- release():
  - P(e);
  - free = true;
  - SIGNAL();

Pair

- Private semaphore b[ID] for each process ID: 0 .. n-1
- Process release is dependent on its location in PAIRS.
- When resource becomes free, the 1\textsuperscript{st} process in line may advance.

PAIRS:

<table>
<thead>
<tr>
<th>ID</th>
<th>P2</th>
<th>P15</th>
<th>P3</th>
<th>P1</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>3</td>
<td>6</td>
<td>17</td>
<td>64</td>
</tr>
</tbody>
</table>

Queue can be ordered according to requested cpu-time

- (requested cpu-time is the resource in this example)
- b[n] = [P1, P3, ... n-1]

Delay

- Place delayed process in queue PAIRS
  - (ordered in ascending requested resource amount order) in correct place
- V(e) – release mutex
- Wait for your turn in private semaphore P(b[ID])
  - Each process has private semaphore, where only that process waits (initial value 0)
  - PAIRS queue determines order, one always wakes up the process at the head of the queue
  - Priority: smallest resource request first

Signal

- If someone waiting, take first one (time, ID), and wake up that process: V(b[ID]);
- o/w V(e)
Semaphore Feature Summary

- Many implementations and semantics
  - Be careful to use
  - E.g., is the (process) scheduler called after each V()?
  - Which one continues with processor, the process executing V() or the process just woken up?
  - Can critical section continue after V()?
  - Busy wait vs. suspend state?

- **Hand coded** synchronization solutions
  - Can solve almost any synchronization problem
  - Baton passing is useful and tricky
  - Explicit handover of some resource
  - Be careful to use
    - Do not leave mutexes open
    - Do not suspend inside mutex
    - Avoid deadlocks
    - Do (multiple) P's and V's in correct order