Lesson 7: Readers and Writers

Semaphore Use In Synchronization
Ch 6 [BenA 06]

Consumer Producer Revisited
Readers and Writers
Baton Passing
Private Semaphores
Resource Management

Lesson 7

Synchronization with Semaphores

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

Process R[i = 1 to 4]
... V(gate); # signal Q
...

Process Q
... P(gate); no need to know nr of R's
... # 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[1]); P(g[2]); P(g[3]); P(g[4]);
... # Q must know number of R's

Barrier Synchronization with Semaphores

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

Process Q[i = 1 to 4]
... V(g[i]); # signal others
P(cont); # wait for others
...

Process Barrier
... P(g[1]); P(g[2]); P(g[3]); P(g[4]); # wait for all
V(cont); V(g[1]); V(g[2]); V(g[3]); V(g[4]); # signal all
... # Barrier must know number of Q's

# Barrier must know number of Q's

Barrier Synchronization with Barrier OS-Primitive

- Specific synchronization primitive in OS
  - Implemented with semaphores...
  - No need for extra process – less process switches?

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

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

Process W
... P(mutex)
write_buffer(data)
V(mutex)
V(data_ready); # signal Q
...

Process R
... P(data_ready); # wait for data
P(mutex)
read_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

Communication with Semaphores Correctly

Sem mutex=1, data_ready = 0, buffer_empty=1;
Int buffer

Process W
... P(buffer_empty);# wait for data
P(mutex)
write_buffer(data)
V(mutex)
V(data_ready); # signal Q
...

Process R
... P(data_ready); # wait for data
P(mutex)
read_buffer(data)
V(mutex)
V(buffer_empty)

# 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

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

Not good
Simultaneously?

Better Solution #2

```
Producer 
process Producer [i=1 to M] {
  while(true) {
    ... produce data …
    P(space); /* wait until space to write*/
    buf[rear] = data; 
    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);
  }
}
Consumer 
process Consumer [i=1 to N] {
  while(true) {
    P(items); /* wait until items to consume */
    data=buf[front]; 
    P(mutex); 
    P(mutex); /* all of this must be in mutex */ 
    if (count == 1) V(items); /* first item to empty buffer */
    if (count < n) V(space); /* still room for next producer */
    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)?

### Solution #3

```c
#define buf[n];      /* n element buffer */
#define front=0, /* read from here */
#define rear=0,          /* write to this one */
#define count=0,       /* nr of items in buf */
#define cwp=0, /* nr of waiting producers */
#define cwc=0 /* nr of waiting consumers */
#define space=1, /* need this to write */
#define items=0, /* need this to read */
#define 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); /* 1st consumer */
    if (count < n && cwp>0) V(space);   /* next producer */
    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);  /* 1st producer */
    if (count > 0 && cwc > 0) V(items); /* next consum */
    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 …)

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

Note:
This is not a critical section (CS) problem, but CS's may be used in the solution.

Requirements
- too complex for plain CS.

Simple (mutex) solution
- Only one reader or writer at a time (not good)

Readers and Writers with Baton
Passing Split Binary Semaphores

- Split 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)

- If reader waiting and no writers: V(r)
  - Do not release mutex (currently reserved e, r, or w)
  - New reader will continue with mutex already locked
- 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
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
```

Still readers first

Unnecessary parts of SIGNAL code was removed

Modify to give writers priority?

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

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
      - int 0 – wait for permission to continue

Simple Very Bad Solution

```c
sem Xmutex = 1, Xavail = N
Xres_request ()  # one unit at a time
P(Xavail)       # ok if always
P(Xmutex)
take 1 unit    # not simple,
V(Xmutex);
Xres_release ()  # one unit at a time
P(Xmutex)
return 1 unit  # may take long time?
V(Xmutex);
V(Xavail);
```

- **What is wrong?**
  - Everything
  - Mutex?
  - Deadlock?
  - Unnecessary delays?
  - Each P() may result in (long) delay?
  - Hold mutex while waiting for resource
  - Very, very bad
  - Others can not get mutex to release resources…

Another Not So Good Solution

```c
sem Xmutex = 1, Xavail = N
Xres_request () # one unit at a time
P(Xmutex)  # ok if always
P(Xmutex)  # allocate unit 1 unit
take 1 unit  # not simple,
V(Xmutex);
Xres_release ()  # one unit at a time
P(Xmutex)
return 1 unit  # may take long time?
V(Xmutex);
V(Xmutex);
```

- **What is wrong?**
  - Works only for resources allocated and freed one unit at a time?
  - Mutex?
    - Mutex of control data?
    - Mutex of resource allocation data structures?

Resource Management with Baton Passing Split Semaphore

- **What is wrong?**
  - Everything
  - Mutex?
  - Deadlock?
  - Unnecessary delays?
  - Each P() may result in (long) delay?
  - Hold mutex while waiting for resource
  - Very, very bad
  - Others can not get mutex to release resources…
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
  - Can create own semaphore arrays 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

Shortest Job Next

(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
  
```c

request(time,ID): # requested time, user id
P(e);
if (!free) BLOCK(); # wait for your turn
free = false;           # got it!
V(e);                   # not SIGNAL(), only 1 at a time

release():
P(e);
free = true;
SIGNAL();         # who gets the next one?
```

Queue can be ordered according to requested cpu-time (requested cpu-time is the resource in this example)

<table>
<thead>
<tr>
<th>ID</th>
<th>P2</th>
<th>P15</th>
<th>P3</th>
<th>P1</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>6</td>
<td>17</td>
<td>64</td>
<td></td>
</tr>
</tbody>
</table>

Process release is dependent on its location in PAIRS. When resource becomes free, the 1\textsuperscript{st} process in line may advance.
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 mutex handover of some type of resource use
    - Private semaphores
      - Explicit signal to some specific process
    - Be careful to use
      - Do not leave mutex’es open, do not suspend inside mutex
      - Avoid deadlocks, do (multiple) P’s and V’s in correct order