Semaphore Use in Synchronization

Ch 6 [BenA 06]

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

Synchronization with Semaphores

\[
\text{sem gate} = -3; \quad \# \text{must know number of R's (!)}
\]

\[
\text{Process R[i = 1 to 4]}
\]

\[
\ldots \quad \text{V(gate)}; \quad \# \text{signal Q}
\]

\[
\ldots
\]

\[
\text{Process Q}
\]

\[
\ldots \quad \text{P(gate)}
\]

\[
\ldots
\]

\[
\# \text{how to prepare for next time?}
\]

\[
\# \text{sem_set (gate, -3) ??}
\]

\[
\text{Process R[i = 1 to 4]}
\]

\[
\ldots \quad \text{V(g[i])}; \quad \# \text{signal Q}
\]

\[
\ldots
\]

\[
\text{Process Q}
\]

\[
\ldots \quad \text{P(g[i])}; \quad \text{P(g[2])}; \quad \text{P(g[3])}; \quad \text{P(g[4])};
\]

\[
\ldots
\]

\[
\# \text{Q must know number of R's}
\]

Barrier Synchronization with Semaphores

\[
\text{sem g[i = 1 to 4] = 0;}
\]

\[
\text{cont} = 0;
\]

\[
\text{Process Q[i = 1 to 4]}
\]

\[
\ldots \quad \text{V(g[i])}; \quad \# \text{signal others}
\]

\[
\text{P(cont) \# \text{wait for others}}
\]

\[
\ldots
\]

\[
\text{Process Barrier}
\]

\[
\ldots \quad \text{P(g[1]); P(g[2]); P(g[3]); P(g[4]); \# \text{wait for all}}
\]

\[
\ldots \quad \text{V(cont); V(cont); V(cont); V(cont); \text{signal all}}
\]

\[
\ldots
\]

\[
\# \text{Barrier must know number of Q's}
\]

Barrier Synchronization with Barrier OS-Primitive

• Specific synchronization primitive in OS
  – Implemented with semaphores…
  – No need for extra process – less process switches

\[
\text{barrier br;}
\]

\[
\text{barrier_init (br, 4); \quad \# \text{must be done before use}}
\]

\[
\text{process Q[i = 1 to 4]}
\]

\[
\ldots \quad \text{barrier_wait (br) \# \text{wait until all have reached this point}}
\]

\[
\text{if (pid==1) \# \text{is this ok? is this done in time?}}
\]

\[
\ldots \quad \text{barrier_init (br, 4) \# \text{must be done before use}}
\]

Communication with Semaphores

\[
\text{Sem mutex}=1, \text{data ready}=0; \quad \text{int buffer} \# \text{one data item buffer}
\]

\[
\text{Process W}
\]

\[
\ldots \quad \text{P(mutex)}
\]

\[
\text{write_buffer(data)}
\]

\[
\text{V(mutex)}
\]

\[
\text{V(data ready); \# \text{signal Q}}
\]

\[
\ldots
\]

\[
\text{Process R}
\]

\[
\ldots \quad \text{P(data ready); \# \text{wait for data}}
\]

\[
\text{P(mutex)}
\]

\[
\text{read_buffer(data)}
\]

\[
\text{V(mutex)}
\]

\[
\text{V(data ready); \# \text{signal Q}}
\]

\[
\ldots
\]

\[
\# \text{What is wrong?}
\]

\[
\text{W might rewrite data buffer before R reads it}
\]

\[
\# \text{Might have extra knowledge to avoid the problem}
\]

Communication with Semaphores Correctly

Fast W can not overtake R now
One reader R, one writer W, binary semaphores
Communication with buffer in shared memory
  – Use: 1 producer – 1 consumer – size 1 buffer
Producer-Consumer with Binary Semaphores

(Liisa Marttinen)

- 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 */)

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) {
        /* consume data */
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);
    }
}

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 */)

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++; /* 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);
    }
}

process Consumer [i=1 to N] {
    while(true) {
        /* consume data */
P(items); /* wait until items to consume */
data=buf[front]; front = (front+1) %n; /* outside mutex, ok? */
P(mutex);
count--; /* this must be in mutex */
if (count == 0) V(space); /* still room for next producer */
if (count > 0) V(items); /* still items for next consumer */
V(mutex);
    }
}
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)?

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
Readers and Writers Problem

- Shared data structure or data base
- 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

Jeff Magee example
(Imperial College, London)

Readers and Writers with Baton Passing

- 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
- In critical control areas only one process advances at a time
  - New reader/writer: wait in P(e)
  - Waiting for read turn: V(e); P(r)
  - Waiting while not holding mutex
    - Waiting for write turn: V(e); P(w)
  - Waiting 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)

SIGNAL()
**Baton Passing with SIGNAL**

```plaintext
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 is removed

Modify to give writers priority?

![Diagram of readers/writers solution using passing the baton (with SIGNAL code)](image)

**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 Bad Solution**

```plaintext
sem Xmutex = 1, Xavail = N

Xres_request ()  // one unit at a time
P(Xmutex)
if "not enough free units" {V(Xmutex); P(Xavail);}
take 1 unit  // assume short time
if "requests pending and enough free units" {V(Xavail);}
else V(Xmutex);
V(Xmutex);

Xres_release ()
P(Xmutex)
return 1 unit
V(Xavail);
V(Xmutex);
```

- 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...

**Resource Management with Baton Passing Split Semaphore**

```plaintext
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
if "requests pending and enough free units" {V(Xavail);}
else V(Xmutex);
V(Xmutex);
Xres_release (K)
P(Xmutex)
return K units
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?
Problems with Resource Management

- Need strong semaphores
- Strong semaphores are FIFO
  - What if 1st in line want 6 units, 2nd 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 1st 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

Private Semaphore Use Example

- Common resource allocation method
  - Here: time = amount of resource requested
  - Here: just select next job (with shortest time)
  - Here: just one job (at most) holding the resource at a time
- Use private semaphores

  ```
  request(time, id) = P(e);
  if (!free)
    DELAY(); # wait for your turn
  free = false;
  SIGNAL();
  V(e);
  ```

- release():

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

  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)
    - PROCESS queue determines order, one always wakes up the process at the head of the queue
      - Priority: smallest resource request first
  - SIGNAL (in Release)
    - If someone waiting, take first one (time, ID), and wake up that process: V(b[ID]);
    - o/w V(e)

PAIRS: P2 P15 P3 P1

<table>
<thead>
<tr>
<th>ID</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>17</td>
<td>54</td>
</tr>
</tbody>
</table>

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

0 1 2 3 n-1

b[n] P1 P3 ...

Private semaphore b[ID] for each process ID: 0 ... n-1

When resource becomes free, the 1st process in line may advance.

CS

```
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?
  - 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 mutex’es open
    - Do not suspend inside mutex
    - Avoid deadlocks
    - Do (multiple) P’s and V’s in correct order