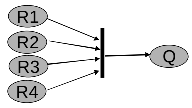
Lesson 7

Semaphore Use In Synchronization

Ch 6 [BenA 06]

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

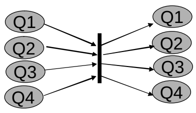
Synchronization with Semaphores




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

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

Process Q
….
P (gate)
…
# 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
```

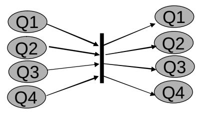

Barrier Synchronization with Semaphores




```
sem g[i = 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(cont); V(cont); V(cont); #signal all
…
# Barrier must know number of Q's
```

- Barrier is implemented as separate *process*
  - This is just one possibility to implement the barrier
  - Cost of process switches?
  - How many process switches?

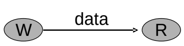
Barrier Synchronization with Barrier OS-Primitive



- Specific synchronization primitive in OS
  - Implemented with semafores…
  - 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)
  …
```

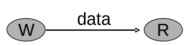

Communication with Semaphores




```
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)
….
```

- What is wrong?
  - W might rewrite data buffer before R reads it
  - Might have extra knowledge to avoid the problem

Communication with Semaphores Correctly




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

Process W
….
P(buffer_empty);
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
- 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 — mutex
- Two important state changes
  - Empty buffer becomes not empty
    - Consumer may need to be awakened — items
  - Full buffer becomes not full
    - Producer may need to be awakened — space

Simple Solution #1

(Producer-Consumer with Binary Semaphores)

```
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 — Not good
    - Producer inserts item to "rear"
    - Consumer removes item from "front" — Simulta-neously?
- First waiting producer/consumer advances when signalled
  - Queued in semaphores

Better Solution #2

(Producer-Consumer with Binary Semaphores)

```
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++;                   /* 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) {
   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)

Ehto-synkro-nointi

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

```
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 */
    cwp=0,         /* nr of waiting producers */
    cwc=0;         /* nr of waiting consumers */
sem space=1,       /* need this to write */
     items=0,      /* need this to read */
     mutex=1;      /* need this to update count */
```


Sol. #3

```
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

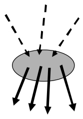
- No simultaneous insert and remove ops
  - Data copying inside critical section
- In general case, only mutex semaphore operations needed
  - Most of the time?
  - Can they be busy-wait semaphores?
- First waiting producer/consumer does not necessarily advance when signalled
  - Someone else may get mutex first
    - E.g., consumer signals (Vspace), another producer gets 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)?

Solutions #1, #2, and #3

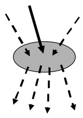
- Which one is best? Why? When?
- How to maximise concurrency?
  - Separate data transfer (insert, remove) from permission to do it
    - Allow obtaining permission (e.g., code with *P(space)* and updating *count*) for one process run concurrently with data transfer for another process (e.g., code with *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)

http://www.doc.ic.ac.uk/~jnm/book/book_applets/ReadersWriters.html


```
sem rw = 1;
process Reader[i = 1 to M] {
  while (true) {
    ...
    P(rw);      # grab exclusive access lock
    read the database;
    V(rw);      # release the lock
  }
}
process Writer[j = 1 to N] {
  while (true) {
    ...
    P(rw);      # grab exclusive access lock
    write the database;
    V(rw);      # release the lock
  }
}
```

reader entry protocol

reader exit protocol

writer entry protocol

writer exit protocol

(Fig 4.8 [Andr00])

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


```
int nr = 0;      # number of active readers
sem rw = 1;      # lock for reader/writer synchronization
process Reader[i = 1 to M] {
  while (true) {
    ...
    ⟨ nr = nr+1;
      if (nr == 1) P(rw);  # if first, get lock
    ⟩
    read the database;
    ⟨ nr = nr-1;
      if (nr == 0) V(rw);  # if last, release lock
    ⟩
  }
}
process Writer[j = 1 to N] {
  while (true) {
    ...
    P(rw);
    write the database;
    V(rw);
  }
}
```

std mutex

Only the first reader waits

Release mutex before P(rw)? (no need)

Writers may starve – not good.
Writers have no chance to cut in between readers.

Jeff Magee example

How should you adjust the readers to starve writers?

(Fig 4.9 [Andr00])

Readers and Writers with Baton Passing Split Binary Semaphore

- Component semaphores e, r, w $0 \le e+r+w \le 1$
  - Mutex wait in P(e), initially 1
  - Readers wait in P(r) if needed, initially 0 (Fig 4.13 [Andr00])
  - Writers wait in P(w) if needed, initially 0 (Alg. 6.21 [BenA06])
- 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

P(e) … V(e)
P(e) … V(r)
P(e) … V(w)
P(r) … V(r)
P(r) … V(w)
…
P(w) … V(e)
…


```
int nr = 0,   ## RW: (nr == 0 or nw == 0) and nw <= 1
    nw = 0;
sem e = 1,    # controls entry to critical sections
    r = 0,    # used to delay readers
    w = 0;    # used to delay writers
              # at all times 0 <= (e+r+w) <= 1
int dr = 0,   # number of delayed readers
    dw = 0;   # number of delayed writers
```

Andrews Fig. 4.12: Outline of readers and writers with passing the baton.

```
process Reader[i = 1 to M] {
  while (true) {
    # ⟨await (nw == 0) nr = nr+1;⟩
      P(e);
      if (nw > 0)
        { dr = dr+1; V(e); P(r); }
      nr = nr+1;
      SIGNAL;
    read the database;
    # ⟨nr = nr-1;⟩
      P(e);
      nr = nr-1;
      SIGNAL;
  }
}
```

```
process Writer[j = 1 to N] {
  while (true) {
    # ⟨await (nr == 0 and nw == 0) nw = nw+1;⟩
      P(e);
      if (nr > 0 or nw > 0)
        { dw = dw+1; V(e); P(w); }
      nw = nw+1;
      SIGNAL;
    write the database;
    # ⟨nw = nw-1;⟩
      P(e);
      nw = nw-1;
      SIGNAL;
  }
}
```

(rStart) (rExit) (wStart) (wExit)

?? P(e) … V(w) ? P(r) … V(w) ? P(e) … V(e) ? P(r) … V(e) ? P(e) … V(w) ?

Baton passing = "do not just release CS, give it to someone special…"

Baton passing

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

SIGNAL()

- 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 "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 (to execute entry or exit protocols)
  - New process gets in mutex only when no old one can be advance
  - Can happen concurrently when reading database

Baton Passing with SIGNAL

SIGNAL – CS baton passing, priority on readers

```
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
```

not possible in wStart, rExit

not possible in rStart

"pass the baton within CS"

"just complete CS"


```
process Reader[i = 1 to M] {
  while (true) {
    # ⟨await (nw == 0) nr = nr+1;⟩
      P(e);
      if (nw > 0) { dr = dr+1; V(e); P(r);}
      nr = nr+1;
      if (dr > 0) { dr = dr-1; V(r);}
      else V(e);
    read the database;
    # ⟨nr = nr-1;⟩
      P(e);
      nr = nr-1;
      if (nr == 0 and dw > 0)
        { dw = dw-1; V(w); }
      else V(e);
  }
}
```

```
process Writer[j = 1 to N] {
  while (true) {
    # ⟨await (nr == 0 and nw == 0) nw = nw+1;⟩
      P(e);
      if (nr > 0 or nw > 0)
        { dw = dw+1; V(e); P(w); }
      nw = nw+1;
      V(e);
    write the database;
    # ⟨nw = nw-1;⟩
      P(e);
      nw = nw-1;
      if (dr > 0) { dr = dr-1; V(r); }
      elseif (dw > 0) { dw = dw-1; V(w); }
      else V(e);
  }
}
```

next reader · 1st reader · 1st writer · next writer

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

Still readers first

Unnecessary parts of *SIGNAL* code is removed

Modify to give writers priority?

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

use printer
use webcam
access database
access CS
allocate memory
allocate buffer
use comm port
get user focus
etc. etc.

Simple Bad Solution

```
sem Xmutex = 1, Xavail = N

Xres_request ()  # one unit at a time
    P(Xmutex)
    P(Xavail)  # ok if always
               # allocate just 1 unit
    take 1 unit  # not simple,
                 # may take long time?
    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…

Another Not So Good Solution

```
sem Xmutex = 1, Xavail = N

Xres_request ()   # one unit at a time
    P(Xavail)     # ok if always
                  # allocate just 1 unit
    P(Xmutex)
    take 1 unit   # not simple,
                  # may take long time?
    V(Xmutex);

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

- 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

sem Xmutex = 1, Xavail = 0 (not N) ; split semaphore
; (short wait) (long wait)

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);

Xres_release (K)
- P(Xmutex)
- return K units
- **if** "requests pending and enough free units" {V(Xavail);}
- **else** V(Xmutex);

CS · CS · baton passing · baton passing

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

Process User

P(mutex)
set up resource demands
V(mutex)
P(me.PrivSem)

Process Server

P(mutex)
locate next process Q to release
V(Q.PrivSem)
V(mutex)

Shortest Job Next (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): # requested time, user id
  P(e);
  if (!free) *DELAY(); # wait for your turn*
  free = false; *# got it!*
  V(e); *# not SIGNAL(), only 1 at a time*
  (CS)
- **release**(): ??
  P(e);
  free = true;
  SIGNAL(); # who gets the next one?
  # pass baton, or release mutex
  (CS)

- 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 (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	ID
3	6	17	64	time

request(26, P11)

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

Process release is dependent on its location in PAIRS. When resource becomes free, the 1st process in line may advance.


```
bool free = true;
sem e = 1, b[n] = ([n] 0);  # for entry and delay
typedef Pairs = set of (int, int);
Pairs pairs = ∅;
## SJN:  pairs is an ordered set ∧ free ⇒ (pairs == ∅)
request(time,id):
  P(e);
  if (!free) {
    insert (time,id) in pairs;
    V(e);            # release entry lock
    P(b[id]);        # wait to be awakened
  }
  free = false;
  V(e);    # optimized since free is false here
release():
  P(e);
  free = true;
  if (P != ∅) {
    remove first pair (time,id) from pairs;
    V(b[id]);  # pass baton to process id
  }
  else V(e);
```

CS

CS

Andr00 Fig. 4.14
Shortest job next (cpu scheduling policy) allocation using semaphores.

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

16.11.2008 37