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) ; 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

Barrier Synchronization with Barrier OS-Primitive

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

• Specific synchronization primitive in OS
  – Implemented with semaphores…
  – No need for extra process – less process switches?
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 that avoids 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
- Actual communication with buffer in shared memory
  – Use model: 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 \textit{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-Consumer with Binary Semaphores)

```c
#include <semaphore.h>

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 */
```
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-Consumer with Binary Semaphores)

```c
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++;                          /* 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);
    }
}

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--;                          /* all of this must be in mutex */
        if (count == n-1) V(space);  /* buffer was full */
        if (count > 0) V(items); /* still items for next consumer */
        V(mutex);
        ... consume data …
    }
}
```

Sol. #2
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
typedef 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 */
```
process Producer \([i=1 \text{ to } M]\) {
    \text{while(true) \{ }
    ... produce data ...
    \text{P(mutex);}
    \text{while (count == n) /* usually not true? \textbf{while, not if}!!!*/}
    \text{cwp++; V(mutex); P(space); P(mutex); cwp--; }
    \text{buf[rear] = data; rear = (rear+1) \% n; count++;}
    \text{if (count == 1 \&\& cwc>0) V(items); /* 1st consumer */}
    \text{if (count < n \&\& cwp>0) V(space); /* next producer */}
    \text{V(mutex);}
    \text{\}}
}

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

\textbf{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 \textbf{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?
    - \textbf{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 \( buf[rear] = data; \ldots \))
    - 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 \ldots\)
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

Note:
This is not a critical section (CS) problem, but CS's may be used in the solution.
Requirements are too complex for plain CS.

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

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

(Fig 4.8 [Andr00])
How should you adjust the readers to not starve writers?

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

std::mutex

Jeff Magee example

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

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

\[
\begin{align*}
\text{P(e)} &\ldots \text{V(e)} \\
\text{P(e)} &\ldots \text{V(r)} \\
\text{P(e)} &\ldots \text{V(w)} \\
\text{P(r)} &\ldots \text{V(r)} \\
\text{P(r)} &\ldots \text{V(w)} \\
\text{P(w)} &\ldots \text{V(e)} \\
\end{align*}
\]
Concurrent Programming (RIO) 6.2.2012

Lecture 7: Readers and Writers 12

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

Baton passing = “do not just release CS or turn, give it to someone special…”

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
      “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

\[\text{SIGNAL} - \text{CS baton passing, priority on readers}\]

\[
\begin{align*}
\text{if (nw == 0 and dr > 0)} & \{ \\
\text{dr = dr -1; } & \text{# wake up waiting reader} \\
\text{V(r); } & \} \\
\text{else if (nr == 0 and nw == 0 and dw > 0)} & \{ \\
\text{dw = dw -1; } & \text{# wake up waiting writer} \\
\text{V(w); } & \} \\
\text{else} & \{ \\
\text{V(e); } & \text{# let new process to mix} \\
\end{align*}
\]

"pass the baton within CS"

"just complete CS"

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
    • init 0 - wait for permission to continue

Simple Very 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

- 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?

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

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);
```

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

\[
\begin{align*}
P(\text{mutex}) & \quad \text{set up resource demands} \\
V(\text{mutex}) & \\
P(\text{me.PrivSem}) &
\end{align*}
\]

Process Server

\[
\begin{align*}
P(\text{mutex}) & \quad \text{locate next process } Q \text{ to release} \\
V(Q\text{.PrivSem}) & \\
V(\text{mutex}) &
\end{align*}
\]
Shortest Job Next
(Private Semaphore Use Example)

- Common resource allocation method
  - Here: \textit{time} = \textit{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

\begin{center}
\begin{tabular}{|c|}
\hline
\textbf{CS} \\
\hline
\{ request(time,id): \# requested time, user id \\
 \quad P(e); \\
 \quad if (!free) \text{DELAY(); \# wait for your turn} \\
 \quad free = false; \quad \# \text{got it!} \\
 \quad V(e); \quad \# \text{not SIGNAL(), only 1 at a time} \\
\hline
\textbf{CS} \\
\hline
\{ release(): \# who gets the next one? \\
 \quad P(e); \\
 \quad free = true; \quad \# \text{pass baton, or release mutex} \\
 \quad SIGNAL(); \\
\hline
\end{tabular}
\end{center}

- \textbf{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
- \textbf{SIGNAL} (in Release)
  - If someone waiting, take first one (time, ID), and wake up that process: V(b[ID]);
  - o/w V(e)
Queue can be ordered according to requested cpu-time (requested cpu-time is the resource in this example)

Private semaphore $b[\text{ID}]$ for each process ID: $0 \ldots n-1$

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 = \emptyset;
## S/N: pairs is an ordered set \land free \Rightarrow (pairs == \emptyset)
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 != \emptyset) {
    remove first pair (time, id) from pairs;
    V(b[id]); # pass baton to process id
} else 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 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