ID1217 Concurrent Programming
Lecture 7

Semaphores (part 1 of 2)

Vladimir Vlassov
KTH/ICT/ECS
VT 2008

Outline (Chapter 4)

- This lecture (Sections 4.1, 4.2)
  - Origin, syntax and semantics of semaphores
  - Types of semaphores: binary, split binary, general.
  - Use of binary semaphores for synchronization
    - Critical sections with semaphores
    - Signaling semaphores
      - Barriers using semaphores
    - Split binary semaphores
      - Consumer-Producer problem
  - Use of general semaphores
    - Bounded buffer problem
- Next lecture (Sections: 4.3-4.5)
  - Selective mutual exclusion using semaphores
  - Resource allocation and scheduling using semaphores
  - Semaphores in Pthreads

Semaphores

- ***Semaphores*** is a unified, low-level but efficient signaling mechanism for both mutual exclusion and condition synchronization.
- Origin of semaphores:
  - Invented in 1968 by Edsger Dijkstra, Dutch computer scientist
  - Inspired by a railroad semaphore – Up/Down signal flag
  - THE system – the first OS used semaphores
  - Semaphore operations
    - P stands for Dutch "Proberen" (to test) or "Passeren" (to pass)
    - V stands for Dutch " Verhogen" (to increment) or "Vrijgeven" (to release)

Syntax and Semantics of Semaphores

- A ***semaphore*** is a special kind of a shared integer variable (or an abstract data type) which can only be accessed using the following two atomic operations:

  pass (proberen) `P(s): < await (s > 0) s = s - 1; >`

  release (verhogen) `V(s): < s = s + 1; >`

  – The value of a semaphore is ***nonnegative***.

  – Declaration and initialization:

```
sem s = expr;                 # single semaphore
sem s[1:n] = ([n] expr)       # array of semaphores
```

    - If not initialized, defaults to zero

Implementation

- Blocking semantics: To avoid busy-waiting, a semaphore should have an associated queue of processes (usually a FIFO).
- The P(s) operation:

```
if (s > 0) s--;
else  { /* wait until the semaphore is up */
```

  Block the proc, i.e. add the proc to the semaphore's queue;

```
}
```

  - The blocked process can be released by V(s);
- The V(s) operation:

```
if (queue is empty) s++;
else { /* "pass the baton " */
```

  Resume one blocked proc, i.e. take the front proc off the semaphore's queue and move it to the ready queue;

```
}
```

Types of Semaphores

- ***Binary semaphore*** takes the value 0 or 1
  - Can be used for mutual exclusion (mutex) and/or condition synchronization (signaling semaphores – much like flags)
- ***Split Binary semaphore*** is a set of binary semaphores where at most one semaphore is 1 at a time:

$$0 \leq s_0 + s_1 + \ldots + s_{n-1} \leq 1$$

  - Can be used for both mutex and condition synchronization
- ***General (counting) semaphore*** takes any nonnegative integer value
  - Can be used for condition synchronization.
  - Serves as a resource counter: counts the number of recourse units

Critical Sections Using a Binary Semaphore (Mutex)

- A critical section of code **< S; >** can be executed with mutual exclusion by enclosing it between P and V operations on a binary semaphore.
  - The mutex semaphore is initialized to 1 to indicate CS is free

```
sem mutex = 1;

process CS[i = 1 to n] {
  while (true) {
    P(mutex);
    critical section;
    V(mutex);
    noncritical section;
  }
}
```

Signaling Semaphores

- A ***signaling semaphore*** is a binary semaphore used for signaling events or conditions.
  - Usually initialized to 0 to indicate absence of an event
  - A process signals an event by V(s) – "sends" a signal
  - A process waits for event by P(s) – "receives" a signal
- Barriers using signaling semaphores
  - Use two binary semaphores per a pair-wise barrier to signal arrival at the barrier and to control departure
  - N-process barrier can be built by combining pair-wise barriers

```
sem arrive1 = 0, arrive2 = 0;
process Worker1 {
  ...
  V(arrive1);     /* signal arrival         */
  P(arrive2);     /* wait for other process */
  ...
}
process Worker2 {
  ...
  V(arrive2);     /* signal arrival         */
  P(arrive1);     /* wait for other process */
  ...
}
```

Use of Binary Semaphores for Synchronization

- Mutual exclusion (atomicity) **<S;>**
  - Use a mutex semaphore:

```
sem mutex = 1; /* CS entry semaphore */
P(mutex);
S;
V(mutex);
```

- Condition synchronization **<await (B) S;>**
  - Use two semaphores: a mutex semaphore for exclusive access and a signaling semaphore to wait for condition

```
sem entry = 1; /* CS entry semaphore */
sem cond = 0;  /* signaling semaphore */
P(entry);
while (!B) { V(entry); P(cond); P(entry);}
S;
V(entry);
```

  - The "***Passing the baton***" technique can be used to optimize the code above.

The "Passing-the-Baton" Technique

- Assume, a proc W awaits for a condition in its CS:

  W: `P(entry); while (!B) { V(entry); P(cond); P(entry);}`

  – To proceed, it needs two semaphores: first **`cond`** and then **`entry`**

- Assume, a proc S sets the condition and signals:

  S: `P(entry); B = true; V(cond); V(entry);`

  – It sets two semaphores: **`cond`** – to signal and **`entry`** – to release CS

- ***The "Passing-the-baton" technique***:

  – A signaling proc (S) passes the entry semaphore to the waiting proc (W) by using the signaling (condition) semaphore

  W: `P(entry); while (!B) { V(entry); P(cond);}`

  S: `P(entry); B = true; V(cond);`

“Passing-the-Baton” Illustrated

```
sem entry = 1,/* controls entry to CS */
    cond = 0; /* to signal the condition */
int dp = 0;   /* counts delayed processes */
process W[i = 0 to n - 1] {
   …
   P(entry);
   if (!B) {
      dp++;
      V(entry);
      P(cond); /* delays on cond */
   }
   S;
   if (dp > 0) {
      dp--;
      V(cond); /* pass the “entry baton” */
   } else V(entry);
   …
}
```

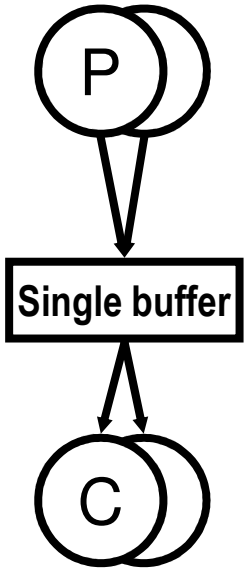
```
process S {
   …
   P(entry);
   change B to true;
   if (dp > 0) {
         dp--;
         V(cond);
   } else V(entry);
   …
}
```

Split Binary Semaphores

- ***A Split Binary semaphore*** is a set of binary semaphores where at most one semaphore is 1 at a time:
  $0 \leq s_0 + s_1 + \ldots + s_{n-1} \leq 1$
  - combines mutual exclusion and signaling;
  - used to indicate alternate states
    - For example: Two binary semaphores **`full`** and **`empty`** forms a split binary semaphore that indicates whether a variable (buffer) is full or empty

Example 1: Consumer/Producer Problem (revisited)

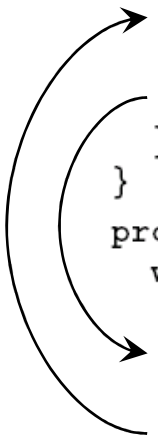
- Illustrates a typical usage of binary semaphores
- Producers (P) and Consumers (C) interact using a single shared buffer.
  - The buffer is assumed to be empty initially.
  - A Producer must be delayed until the buffer is empty
  - A Consumer must be delayed until the buffer is full.
- Solution:
  - Mutual exclusive access to the buffer
  - Condition synchronization
    - C waits for data in the buffer, i.e waits until the buffer is full
    - P waits for place in the buffer, i.e. waits until the buffer is empty
  - Use of two binary semaphores allows to achieve both, mutual exclusion and condition synchronization



Producers and Consumers Using Binary Semaphores for Synchronization

- Two binary semaphores, **`full`** and **`empty`**, form a split binary semaphore.
- Either of them or non of them is 1 at a time:

  $0 \leq$ **`full`** + **`empty`** ≤ 1
- When both are 0 – the buffer is locked and accessed by only one process



```
typeT buf;      /* a buffer of some type T */
sem empty = 1, full = 0;
process Producer[i = 1 to M] {
  while (true) {
    ...
    /* produce data, then deposit it in the buffer */
    P(empty);
    buf = data;
    V(full);
  }
}
process Consumer[j = 1 to N] {
  while (true) {
    /* fetch result, then consume it */
    P(full);
    result = buf;
    V(empty);
    ...
  }
}
```

Property of a Split Binary Semaphore

- Suppose,
  - Processes are using a split binary semaphore;
  - Every execution paths in every proc starts with a P and ends with a V.
- Then all statements between any P and the next V execute with mutual exclusion.
  - By definition: $0 \leq s_0 + s_1 + \ldots + s_{n-1} \leq 1$
  - Observation: wherever any process is between any P and the next V, the semaphores are all zero
  - This implies that no any other process can complete a P until the first process executes a V

General Semaphores

- Can be associated with a shared recourse and serve as a resource counter:
  - Initialized to some integer value – the total amount of recourse units available;
  - Takes any nonnegative integer value – current amount of resource units available;
  - Used for condition synchronization, e.g. wait until a unit of resource is available and can be occupied.

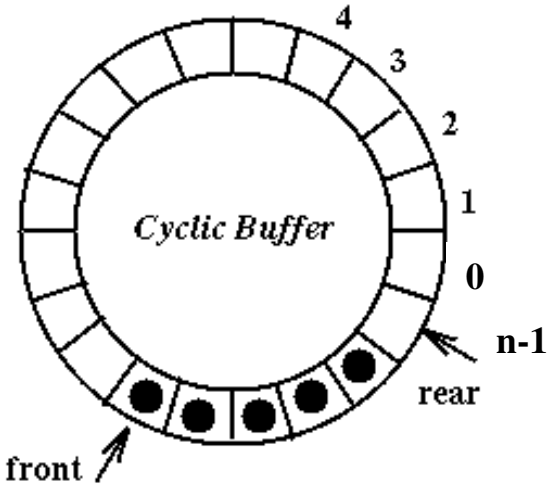
Example 2: Bounded Buffer Problem

- Producers and Consumers interact using a ***bounded buffer*** – a multi-slot communication buffer limited in size (no underflow, no overflow)
- Buffer of size **`n`** as an array of some type *T*:

```
TypeT buf[n];
int front = 0, rear = 0;
```

- Accessing buffer
  - **`rear`** points to an empty slot
  - **`front`** points to the head data item

```
deposit: buf[rear] = data;
         rear = (rear + 1) % n;
fetch:   result = buf[front];
         front = (front + 1) % n;
```



Bounded Buffer Using General Semaphores

- Synchronized access to the bounded buffer:
  - P must wait until there is an empty slot in the buffer
  - C must wait until there is a full slot in the buffer.
- Use two general semaphores:

  `sem empty = n;`

  `sem full = 0;`
  - **`empty`** counts empty slots,
  - **`full`** counts full slots.

```
typeT buf[n];       /* an array of some type T */
int front = 0, rear = 0;
sem empty = n, full = 0;    /* n-2 <= empty+full <= n */
process Producer {
  while (true) {
    ...
    produce message data and deposit it in the buffer;
    P(empty);
    buf[rear] = data; rear = (rear+1) % n;
    V(full);
  }
}
process Consumer {
  while (true) {
    fetch message result and consume it;
    P(full);
    result = buf[front]; front = (front+1) % n;
    V(empty);
    ...
  }
}
```

Example 3: Multiple Producers/Multiple Consumers and a Bounded Buffer

- Assume multiple Producers and multiple Consumers
  - Depositing to the buffer becomes critical section for Producers:

```
buf[rear] = data;
rear = (rear + 1) % n;
```

  - Fetching from the buffer becomes critical section for Consumers:

```
result = buf[front];
front = (front + 1) % n;
```

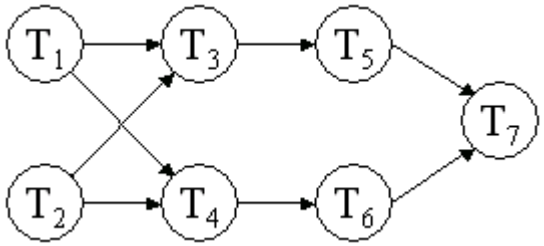
  - Deposit and fetch are not critical to each other because they access different shared locations: deposit uses rear, fetch – front
- To achieve mutual exclusion, use two binary semaphores
  - **`mutexD`** to protect **`rear`** in deposit called by Producers
  - **`mutexF`** to protect **`front`** in fetch called by Consumers

Producers/Consumers Using Semaphores

```
typeT buf[n];       /* an array of some type T */
int front = 0, rear = 0;
sem empty = n, full = 0;      /* n-2 <= empty+full <= n */
sem mutexD = 1, mutexF = 1;   /* for mutual exclusion   */
process Producer[i = 1 to M] {
  while (true) {
    ...
    produce message data and deposit it in the buffer;
    P(empty);
    P(mutexD);
    buf[rear] = data; rear = (rear+1) % n;
    V(mutexD);
    V(full);
  }
}
process Consumer[j = 1 to N] {
  while (true) {
    fetch message result and consume it;
    P(full);
    P(mutexF);
    result = buf[front]; front = (front+1) % n;
    V(mutexF);
    V(empty);
    ...
  }
}
```

Exercise: An Execution Order

- Consider the following precedence graph of processes:



- Each process has the following code outline:

```
process Ti (i = 1, ..., 7) {
        wait for predecessors, if any;
        execute the task;
        signal successor, if any;
   }
```

- For example, in the graph above task T5 has to wait ofr task T3 and signals task T7.
- Develop an implementation of the wait and signal code for each of the seven tasks in the above graph. Use semaphores for synchronization. Try to use a minimal number of semaphores.

Possible Solutions

- Using 5 semaphores, one per edge target:

```
sem S[3:7] = ([5] 0); // init
T1:: ... V(S[3]); V(S[4]);
T2:: ... V(S[3]); V(S[4]);
T3:: P(S[3]); P(S[3])... V(S[5]);
T4:: P(S[4]); P(S[4])... V(S[6]);
T5:: P(S[5]);... V(S[7]);
T6:: P(S[6]);... V(S[7]);
T7:: P(S[7]); P(S[7]);...
```

- As T7 executes after T3 (and T4), one semaphore, e.g. S, e.g. S3, can be reused. One of possible solutions using 4 semaphores:

```
sem S[3:6] = ([4] 0); // init
T1:: ... V(S[3]); V(S[4]);
T2:: ... V(S[3]); V(S[4]);
T3:: P(S[3]); P(S[3])... V(S[5]); V(S[5]);
T4:: P(S[4]); P(S[4])... V(S[6]);
T5:: P(S[5]);... V(S[3]);
T6:: P(S[6]);... V(S[3]);
T7:: P(S[5]); P(S[3]); P(S[3]);...
```