More on Semaphores, and Classic Synchronization Problems

CSCI 3753 Operating Systems
Spring 2005
Prof. Rick Han
Announcements

• HW #3 is due Friday Feb. 25, a week+ from now
• PA #2 is coming, assigned about next Tuesday
• Midterm is tentatively Thursday March 10
• Read chapters 8 and 9
From last time...

- We introduced critical sections, atomic locks, and semaphores
- A semaphore $S$ is an integer variable
  - initialize $S$ to some value $S_{\text{init}} = n$
  - $P(S)$ operates atomically on $S$ to decrement $S$, but only if $S > 0$. Otherwise, the process calling $P(S)$ relinquishes control.
  - $V(S)$ atomically increments $S$
- A binary semaphore, a.k.a. mutex lock, can be used to provide mutual exclusion on critical sections
  - $S_{\text{init}} = 1$
  - value of semaphore varies only between 0 and 1
Semaphores

- Usage example #1: mutual exclusion

Semaphore $S = 1$; // initial value of semaphore is 1
int counter;       // assume counter is set correctly somewhere in code

Process P1:

P($S$);
// execute critical section
counter++;
V($S$);

Process P2:

P($S$);
// execute critical section
counter--;
V($S$);

- Both processes atomically $P()$ and $V()$ the semaphore $S$, which enables mutual exclusion on critical section code, in this case protecting access to the shared variable counter
Semaphores

• Usage example #2: enforcing order of access between two processes
  – Suppose there are two processes P1 and P2, where P1 contains code C1 and P2 contains code C2
  – Want to ensure that code C1 executes before code C2
  – Use semaphores to synchronize the order of execution of the two processes

Semaphore S=0;  // initial value of semaphore = 0

Process P1:
C1;            // execute C1
signal(S);     // V() the semaphore

Process P2:
wait(S);       // P() the semaphore
C2;            // execute C2
Semaphores

• In the previous example #2, there are two cases:
  1. If P1 executes first, then
     • C1 will execute first, then P1 will V() the semaphore, increasing its value to 1
     • Later, when P2 executes, it will call wait(S), which will decrement the semaphore to 0 followed by execution of C2
     • Thus C1 executes before C2
  2. If P2 executes first, then
     • P2 will block on the semaphore, which is equal to 0, so that C2 will not be executed yet
     • Later, when P1 executes, it will run through C1, then V() the semaphore
     • This awakens P2, which then executes C2
     • Thus C1 executes before C2
Semaphores

- Let’s revisit the following intuitive implementation of semaphores that uses only disabling and reenabling of interrupts
  - Note that a process that blocks on this kind of semaphore will spin in a busy wait while() loop - this type of semaphore is called a spinlock
- Figure 8.25 in the text illustrates a semaphore implemented using a TestandSet instruction that also exhibits spinlock behavior

```c
P(S) {
    disableInt();
    while(S==0) {
        enableInt();
        disableInt();
    }
    enableInt();
    S--;
    enableInt()
}

V(S) {
    disableInt();
    S++;
    enableInt()
}
```
Semaphores

• Spinlock implementations of semaphores can occupy the CPU unnecessarily
• Instead, sleep the process until it needs to be woken up by a \texttt{V()}/\texttt{signal()}

\begin{verbatim}
P(semaphore *S) {
    S->value--;  
    if (S->value<0) {
        add this process to S->list;  
        block();
    }
}
\end{verbatim}

\begin{verbatim}
V(semaphore *S) {
    S->value++;  
    if (S->value<=0) {
        remove a process P from S->list;  
        wakeup(P);
    }
}
\end{verbatim}

where we have defined the following structure for a semaphore
typedef struct {
    int value;
    struct process *list;
} semaphore;
Semaphores

• In the previous slide’s redefinition of a semaphore, we are departing from the classic definition of a semaphore
  – Now, the semaphore’s value is allowed to be negative, because the decrement occurs before the test in P()
  – The absolute value of the semaphore’s negative amount can now be used to indicate the number of processes blocked on the semaphore
  – Processes now yield the CPU if the semaphore’s value is negative, rather than busy wait
  – If more than one process is blocked on a semaphore, then use a FIFO queue to select the next process to wake up when a semaphore is V’ed
    • Why is LIFO to be avoided?
Deadlock

• Semaphores provide synchronization, but can introduce more complicated higher level problems like *deadlock*
  – two processes deadlock when each wants a resource that has been locked by the other process
  – e.g. P1 wants resource R2 locked by process P2 with semaphore S2, while P2 wants resource R1 locked by process P1 with semaphore S1
Deadlock

Semaphore Q = 1;  // binary semaphore as a mutex lock
Semaphore S = 1;  // binary semaphore as a mutex lock

Process P1:

P(S);
(1)
P(Q);
(3)
modify R1 and R2;
V(S);
V(Q);

Process P2:

(2) P(Q);
(4) P(S);
Deadlock!
modify R1 and R2;
V(Q);
V(S);

If statements (1) through (4) are executed in that order, then P1 and P2 will be deadlocked after statement (4) - verify this for yourself by stepping thru the semaphore values.
Deadlock

• In the previous example,
  – Each process will sleep on the other process’s semaphore
  – the V() signalling statements will never get executed, so there is no way to wake up the two processes from within those two processes
  – there is no rule prohibiting an application programmer from P()’ing Q before S, or vice versa - the application programmer won’t have enough information to decide on the proper order
  – in general, with N processes sharing N semaphores, the potential for deadlock grows
Deadlock

- Other examples:
  - A programmer mistakenly follows a P() with a second P() instead of a V(), e.g.
    
    \[
    \begin{align*}
    & P(\text{mutex}) \\
    & \text{critical section} \\
    & P(\text{mutex}) \quad \text{<---- this causes a deadlock, should have been a V()} \\
    \end{align*}
    \]
  - A programmer forgets and omits the P(mutex) or V(mutex). Can cause deadlock if V(mutex) is omitted. Can violate mutual exclusion if P(mutex) is omitted.
  - A programmer reverses the order of P() and V(), e.g.
    
    \[
    \begin{align*}
    & V(\text{mutex}) \\
    & \text{critical section} \quad \text{<---- this violates mutual exclusion, but is not an example of deadlock} \\
    & P(\text{mutex}) \\
    \end{align*}
    \]
Classic Synchronization Problems

• Bounded Buffer Producer-Consumer Problem
• Readers-Writers Problem
  – First Readers Problem
• Dining Philosophers Problem
• These are not just abstract problems
  – They are representative of several classes of synchronization problems commonly encountered when trying to synchronize access to shared resources among multiple processes
Bounded Buffer
Producer/Consumer Problem

- Pool of \( n \) buffers, each capable of holding 1 item
- producer takes an empty buffer and produces a full buffer
- consumer takes a full buffer and produces an empty buffer

Graphic © Pearson textbook slides
Bounded Buffer Producer/Consumer Problem

• Synchronization setup:
  – Use a mutex semaphore to protect access to buffer manipulation, $\text{mutex}_{\text{init}} = 1$
  – Use two counting semaphores full and empty to keep track of the number of full and empty buffers, where the values of $\text{full} + \text{empty} = n$
    • $\text{full}_{\text{init}} = 0$
    • $\text{empty}_{\text{init}} = n$
Bounded Buffer
Producer/Consumer Problem

• Why do we need counting semaphores? Why do we need two of them? Consider the following:

Producer:

while(1) {
    // need code here to keep track of number of empty buffers
    P(mutex)
    obtain empty buffer and add next item, creating a full buffer
    V(mutex)
    ...
}

Consumer:

while(1) {
    ...
    P(mutex)
    remove item from full buffer, create an empty buffer
    V(mutex)
    ...
}
Bounded Buffer
Producer/Consumer Problem

- If we add an empty counting semaphore initialized to n, then a producer calling P(empty) will keep decrementing until 0, replacing n empty buffers with n full ones. If the producer tries to produce any more, P(empty) blocks producer until more empty buffers are available - this is the proper behavior that we want.

Producer:

while(1) {
    P(empty)
    P(mutex)
    obtain empty buffer and add next item, creating a full buffer
    V(mutex)
    ...
}

Consumer:

while(1) {
    ...
    P(mutex)
    remove item from full buffer, create an empty buffer
    V(mutex)
    ...
}
Bounded Buffer
Producer/Consumer Problem

- We also need to add \( V(\text{empty}) \), so that when the consumer is done reading, more empty buffers are produced.
- Unfortunately, this solution does not prevent a consumer from reading even when there are no buffers to read. For example, if the first consumer reads before the first producer executes, then this solution will not work.

**Producer:**
```
while(1) {
    P(\text{empty})
    P(\text{mutex})
    \text{obtain empty buffer and add next item, creating a full buffer}
    V(\text{mutex})
    ...
}
```

**Consumer:**
```
while(1) {
    ...
    P(\text{mutex})
    \text{remove item from full buffer, create an empty buffer}
    V(\text{mutex})
    V(\text{empty})
    ...
}
```
Bounded Buffer
Producer/Consumer Problem

• So add a second counting semaphore full, initially set to 0
• Verify for yourself that this solution won’t deadlock, synchronizes properly with mutual exclusion, prevents a producer from writing to a full buffer, and prevents a consumer from reading from an empty buffer

**Producer:**

```c
while(1) {
    P(empty)
    P(mutex)
    P(mutex)
    obtain empty buffer and add next item, creating a full buffer
    V(mutex)
    V(full)
}
```

**Consumer:**

```c
while(1) {
    P(full)
    P(mutex)
    remove item from full buffer, create an empty buffer
    V(mutex)
    V(empty)
}
```
The Readers/Writers Problem

- There are several writer processes that want to write to a shared object, e.g. a file, and also several reader processes that want to read from the same shared object.
- Want to synchronize access

The “First Readers/Writers Problem”: no reader is kept waiting unless a writer already has seized the shared object. We will implement this in the next slides.
The Readers/Writers Problem

• readers share data structures:
  – semaphore mutex, wrt;  // initialized to 1
  – int readcount;  // initialized to 0, controlled by mutex
• writers also share semaphore wrt

Writer:

```c
while(1) {
  wait(wrt);
  // writing
  signal(wrt);
}
```

Reader:

```c
while(1) {
  wait(mutex);
  readcount++;
  if (readcount==1) wait(wrt);
  signal(mutex);

  // reading

  wait(mutex);
  readcount--;  
  if (readcount==0) signal(wrt);
  signal(mutex);
}
```
The Readers/Writers Problem

- If multiple writers seek to write, then the write semaphore `wrt` provides mutual exclusion.
- If the 1st reader tries to read while a writer is writing, then the 1st reader blocks on `wrt`.
  - If subsequent readers try to read while a writer is writing, they block on `mutex`.
- If the 1st reader reads and there are no writers, then 1st reader grabs the write lock and continues reading, eventually releasing the write lock when done reading.
  - If a writer tries to write while the 1st reader is reading, then the writer blocks on the write lock `wrt`.
  - If a 2nd or any subsequent reader tries to read while the 1st reader is reading, then it falls through and is allowed to read.