CS 4410
Operating Systems

Synchronization
Classic Problems

Summer 2013
Cornell University
Today

• What practical problems can we solve with semaphores?
• Bounded-Buffer Problem
• Producer-Consumer Problem
Producer-Consumer Problem

- Arises when **two or more threads communicate** with each other.
- And, some threads “**produce**” data and other threads “**consume**” this data.
- Real example: Production line
Producer-Consumer Problem

- Start by imagining an unbounded (infinite) buffer
  - Producer process writes data to buffer
    - Writes to In and moves rightwards
  - Consumer process reads data from buffer
    - Reads from Out and moves rightwards
    - Should not try to consume if there is no data
Producer-Consumer Problem

- Bounded buffer: size ‘N’
  - Access entry 0… N-1, then “wrap around” to 0 again
- Producer process writes data to buffer
  - Must not write more than ‘N’ items more than consumer “ate”
- Consumer process reads data from buffer
  - Should not try to consume if there is no data
Producer-Consumer Problem

• Multiple producer-threads.
• Multiple consumer-threads.
• One bounded buffer with N entries.
• All threads modify the same buffer.
• Requirements:
  • No production when all N entries are full.
  • No consumption when no entry is full.
  • Only one thread should modify the buffer at any time.
Producer-Consumer Problem

• Solving with semaphores:
  • We’ll use *counters* to track how much data is in the buffer
    - One counter counts as we add data and stops a producer if there are $N$ objects in the buffer.
    - A second counter counts as we remove data and stops a consumer if there are 0 in the buffer.
  • Idea: since general semaphores can count for us, we don’t need a separate counter variable.
  • We'll use a mutex to protect the update of the buffer ("In" and "Out" pointers).
Producer-Consumer Problem

Shared pointers: “In”, “Out”  
Shared Semaphores: mutex, empty, full;

mutex = 1; /* for mutual exclusion*/  
empty = N; /* number empty buf entries */  
full = 0; /* number full buf entries */

**Producer**
do {
    //produce item
    //update “In”
} while (true);

**Consumer**
do {
    //consume item
    //update “Out”
} while (true);
Producer-Consumer Problem

Shared pointers: “In”, “Out”
Shared Semaphores: mutex, empty, full;

\[
\begin{align*}
\text{mutex} &= 1; \quad /* \text{for mutual exclusion} */ \\
\text{empty} &= N; \quad /* \text{number empty buf entries} */ \\
\text{full} &= 0; \quad /* \text{number full buf entries} */
\end{align*}
\]

**Producer**
do {
    wait(empty);
    // produce item
    // update “In”
    signal(full);
} while (true);

**Consumer**
do {
    wait(full);
    // consume item
    // update “Out”
    signal(empty);
} while (true);
Producer-Consumer Problem

Shared pointers: “In”, “Out”
Shared Semaphores: mutex, empty, full;

```
mutex = 1; /* for mutual exclusion*/
empty = N; /* number empty buf entries */
full = 0; /* number full buf entries */
```

**Producer**
do {
    wait(empty);
    wait(mutex);
    //produce item
    //update “In”
    signal(mutex);
    signal(full);
} while (true);

**Consumer**
do {
    wait(full);
    wait(mutex);
    //consume item
    //update “Out”
    signal(mutex);
    signal(empty);
} while (true);
Readers and Writers

• In this problem, threads share data that some threads “read” and other threads “write”.

• Goal: allow multiple concurrent readers but only a single writer at a time, and if a writer is active, readers wait for it to finish.
Readers-Writers Problem

• Access to a database
  • A **reader** is a thread that needs to look at the database but won’t change it.
  • A **writer** is a thread that modifies the database.

• Making an airline reservation
  • When you browse to look at flight schedules the web site is acting as a reader on your behalf.
  • When you reserve a seat, the web site has to write into the database to make the reservation.
Readers-Writers Problem

- Many reader-threads.
- Many writer-threads.
- One piece of data.
- Multiple threads try to access that data.

Requirements:
- Multiple readers may access the data at the same time.
- If a writer accesses the data, no other thread may access the data.

What happens when multiple readers and one writer are waiting to access the data?
Readers-Writers Problem

mutex = Semaphore(1)
wrt = Semaphore(1)
readcount = 0;

Reader
do{

}

Writer
dofor /*writing is performed*/

}while(true)

/*reading is performed*/

}while(true)
Readers-Writers Problem

mutex = Semaphore(1)
wrt = Semaphore(1)
readcount = 0;

**Writer**
do{
    wait(wrt);
    /*writing is performed*/
    signal(wrt);
}while(true)

**Reader**
do{
    wait(wrt);
    /*reading is performed*/
    signal(wrt);
}while(true)
Readers-Writers Problem

mutex = Semaphore(1)
wrt = Semaphore(1)
readcount = 0;

**Writer**
do{
    wait(wrt);
    /*writing is performed*/
    signal(wrt);
}while(true)

**Reader**
do{
    wait(mutex);
    readcount++;  
    if (readcount == 1)
        wait(wrt);
    signal(mutex);
    /*reading is performed*/
    wait(mutex);
    readcount--;
    if (readcount == 0)
        signal(wrt);
    signal(mutex);
}while(true)
Readers-Writers Notes

- If there is a writer
  - First reader blocks on \texttt{wrl}
  - Other readers block on \texttt{mutex}
- Once a reader is active, all readers get to go through
  - Which reader gets in first?
- The last reader to exit signals a writer
  - If no writer, then readers can continue
- If readers and writers are waiting on \texttt{wrl}, and writer exits
  - Who gets to go in first?
- Why doesn’t a writer need to use \texttt{mutex}?
- Is the previous solution fair?
- Readers can “starve” writers!
- Building a “fair” solution is tricky!
Today

- Which practical problems can we solve with semaphores?
- Producers-Consumers Problem
- Readers-Writers Problem