CMSC 321: Operating Systems

Lecture 6

More Concurrency
Semaphores - Recap

- special variable that counts # of signals on it
  - may be initialized to > 0
  - \( P() \) decrements; if < 0, invoking process blocks
  - \( V() \) increments; if \( \leq 0 \), pick a process to unblock

- \( P() \) and \( V() \) assumed atomic
- Queue for processes waiting on semaphore

- For mutual exclusion, initialize semaphore to 1
Semaphores

*Example use of semaphore s:*

\[
P(s);
\]

/* CRITICAL SECTION */

\[
V(s);
\]

*Value (magnitude) of semaphore:*

- \( \geq 0 \): # of processes that can proceed w/o blocking
- \(< 0\): # of processes blocked

*Can’t actually look at the value*

- \(P()\) and \(V()\) are the only methods available

*See Nachos: threads/sync.h threads/sync.cc*

- a bit different...
Semaphore Recap

• Semaphores:
  - use to achieve mutual exclusion
  - use for synchronization

  - low-level implementation (probably with TSL)
    • see Section 6.4 – 6.5 in Silberschatz

  - must develop mutex & synchro solutions yourself
Producer/Consumer Problem

- Two processes share common, infinite buffer

- **Producer**: puts info into buffer
- **Consumer**: takes info from buffer

- Problem: if consumer takes from an empty buffer
Basic (Unprotected) Code

```plaintext
producer() {
    while (true) {
        item = produceItem();
        b[in++] = item;    /* access to buf */
    }
}
}

consumer() {
    while (true) {
        item = b[out++];  /* access to buf */
        consumeItem(item);
    }
}
```
Semaphores for Producer/Consumer

• Must control access to shared buffer

• Can use semaphore(s)...
  – but how many?
  – for what purpose?

• What if multiple producers?
A Semaphore-Based Solution

/* semaphores defined here */

producer() {
    while (true) {
        item = produceItem();
        b[in++] = item;
    }
}

consumer() {
    while (true) {
        item = b[out++];
        consumeItem(item);
    }
}
A Semaphore-Based Solution

Semaphore access = 1;
Semaphore itemsAvailable = 0;

producer() {
    while (true) {
        item = produceItem();
P(access);
b[in++] = item;
V(access);
V(itemsAvailable);
    }
}

consumer() {
    while (true) {
P(itemsAvailable);
P(access);
item = b[out++];
V(access);
consumeItem(item);
    }
}
A Semaphore-Based Solution

Semaphore access = 1;
Semaphore itemsAvailable = 0;

**access** is used for mutex

```java
producer() {
    while (true) {
        item = produceItem();
        P(access);
        b[in++] = item;
        V(access);
        V(itemsAvailable);
    }
}

consumer() {
    while (true) {
        P(itemsAvailable);
        P(access);
        item = b[out++];
        V(access);
        V(access);
        consumeItem(item);
    }
}
```
A Semaphore-Based Solution

Semaphore access = 1;
Semaphore itemsAvailable = 0;

itemsAvailable is used for synchronization

producer() {
    while (true) {
        item = produceItem();
P(access);
b[in++] = item;
V(access);
V(itemsAvailable);
    }
}

consumer() {
    while (true) {
        P(itemsAvailable);
P(access);
item = b[out++];
V(access);
consumeItem(item);
    }
}
What if We Swap?

Semaphore access = 1;
Semaphore itemsAvailable = 0;

producer() {
    while (true) {
        item = produceItem();
        P(access);
        b[in++] = item;
        V(access);
        V(itemsAvailable);
    }
}

c Consumer() {
    while (true) {
        P(itemsAvailable);
        P(access);
        item = b[out++];
        V(access);
        V(access);
        consumeItem(item);
    }
}
Order of Operations

Semaphore access = 1;
Semaphore itemsAvailable = 0;

producer() {
  while (true) {
    item = produceItem();
    P(access);
    b[in++] = item;
    V(itemsAvailable);
    V(access);
  }
}

cconsumer() {
  while (true) {
    P(itemsAvailable);
    P(access);
    item = b[out++];
    V(access);
    consumeItem(item);
    V(access);
  }
}

Solution will still work
What if We Swap?

Semaphore access = 1;
Semaphore itemsAvailable = 0;

producer() {
    while (true) {
        item = produceItem();
        P(access);
        b[in++] = item;
        V(access);
        V(itemsAvailable);
    }
}

c consumer() {
    while (true) {
        item = b[out++];
        P(access);
        item = b[out++];
        P(access);
        consumeItem(item);
        V(access);
    }
}
Order of Operations

Semaphore access = 1;
Semaphore itemsAvailable = 0;

producer() {
    while (true) {
        item = produceItem();
        P(access);
        P(access);
        b[in++] = item;
        V(access);
        V(access);
        V(itemsAvailable);
    }
}

c consumer() {
    while (true) {
        P(access);
        P(itemsAvailable);
        item = b[out++];
        V(access);
        V(access);
        consumeItem(item);
    }
}

Can cause deadlock... why?
Notes About Our Solution

• Tanenbaum: “With semaphores, interprocess communication looks easy, right? Forget it.”

• Semaphores: two different purposes
  – access: for mutual exclusion
  – itemsAvailable: for synchronization

• access is a binary semaphore:
  – values 0 and 1 only
  – initialized to 1
  – used to ensure only one process is in its CS

• Can be shown that binary semaphore has same expressive power as counting semaphore
Nachos Project 1

- Implement *locks* and *condition variables*
  - use semaphores, or
  - use `Thread::Sleep()`

- Lock:
  - two states: **BUSY** or **FREE**
  - two ops: `Acquire()` and `Release()`
  - Nachos: `threads/sync.{h,cc}`

- Condition Variable:
  - allows processes/threads to block
  - no associated value
  - must have previously acquired a lock
  - three ops: `Wait()` and `Signal()` and `Broadcast()`

  - originated in *monitors* (more later)
  - signal semantics: Hoare, Brinch Hansen, Mesa (Nachos)
wait() semantics

- Call to wait() releases lock and immediately puts calling thread to sleep
- When does wait() return?
- "Mesa" semantics: wait() must reacquire the lock before returning.
Example: Readers/Writers

- Multiple threads share an area of memory that is both readable and writeable
- Any number of threads can read simultaneously
- Only one thread at a time can write
- No reading allowed when writing
- Writing has precedence over reading.
lock *l = new Lock(""");
Condition readersDone("");
Condition writerDone ("");
bool readyToWrite = False;
int readerCount = 0;

// Shared variables

// Writer
l->acquire();
readyToWrite = True;
while(readerCount > 0)
    readersDone.wait(l);
write();
readyToWrite = False;
writerDone.broadcast(l);
l->release();

// Reader
l->acquire();
while(readyToWrite)
    writerDone.wait(l);
readerCount++;
l->release();
read();
l->acquire();
readerCount--;
if (readerCount == 0)
    readersDone.signal(l);
l->release();
Scenario: one reader reading, writer becomes ready.
readerCount == 1

// Writer
l->acquire();
readyToWrite = True;
while(readerCount > 0)
    readersDone.wait(l);
write();
readyToWrite = False;
writerDone.broadcast(l);
l->release();

// Reader
l->acquire();
while(readyToWrite)
    writerDone.wait(l);
readerCount++;
l->release();
read();
l->acquire();
readerCount--;
if (readerCount == 0)
    readersDone.signal(l);
l->release();
Writer waits
Reader finishes reading

// Writer
l->acquire();
readyToWrite = True;

while(readerCount > 0)
    readersDone.wait(l);
write();

readyToWrite = False;
writerDone.broadcast(l);
l->release();

// Reader
l->acquire();
while(readyToWrite)
    writerDone.wait(l);

readerCount++;
read();

l->acquire();
readerCount--;
if (readerCount == 0)
    readersDone.signal(l);
l->release();

l->release();
Back to mutual exclusion

- Spin locks (busy waiting)
- Uses strict alternation
  - one process could not enter CS twice in a row
  - speed dictated by slowest process

```c
Process P0:
while (TRUE) {
    while (turn != 0) {
        /* sit&spin */;
    }
    /* CS */
    turn = 1;
    /* NON-CS */
}

Process P1:
while (TRUE) {
    while (turn != 1) {
        /* sit&spin */;
    }
    /* CS */
    turn = 0;
    /* NON-CS */
}
```
A Correct Mutex Solution

- Dekker was first to give correct solution
  - software solution w/o strict alternation
  - take turns, lock variables, warning variables

- Peterson (1981) discovered simpler solution
  - algorithm on next slide

- Does not require strict alternation
- Still uses busy waiting
What happens if both processes set turn almost simultaneously?

```
#define FALSE 0  
#define TRUE  1  
#define N      2  /* number of processes */

int turn;     /* whose turn is it? */
int interested[N]; /* all values initially 0 (FALSE) */

void enter_region(int process); /* process is 0 or 1 */
{
    int other;  /* number of the other process */

    other = 1 - process; /* the opposite of process */
    interested[process] = TRUE; /* show that you are interested */
    turn = process;         /* set flag */
    while (turn == process && interested[other] == TRUE) /* null statement */;
}

void leave_region(int process) /* process: who is leaving */
{
    interested[process] = FALSE; /* indicate departure from critical region */
}
```