CMSC 321: Operating Systems

Lecture 8

Famous IPC Problems, Monitors
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

Note: shaded area indicates portion of buffer that is occupied
Basic (Unprotected) Code

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

consumer() {
    while (true) {
        item = b[out++];  /* access to buf */
        consumeItem item);
    }
```
Producer/Consumer: Fixed-Size Buffer

- Two processes share common, fixed-size buffer
- Producer: info into buffer
- Consumer: info from buffer

Problems:
- if producer tries to put into a full buffer
- if consumer tries to take from an empty buffer
Basic (Unprotected) Code

```java
producer() {
    while (true) {
        item = produceItem();
        b[in] = item; /* access to buf */
        in = (in + 1) % SIZE;
    }
}

consumer() {
    while (true) {
        item = b[out]; /* access to buf */
        out = out + 1 % SIZE;
        consumeItem(item);
    }
}
```
Semaphore-Based Solution

/* semaphores defined here */

producer() {
    while (true) {
        item = produceItem();
        b[in] = item;
        in = (in + 1) % SIZE;
    }
}

consumer() {
    while (true) {
        item = b[out];
        out = (out + 1) % SIZE;
        consumeItem(item);
    }
}
Semaphore-Based Solution

Semaphore access = 1;
Semaphore spacesForProducer = SIZE;
Semaphore itemsForForConsumer = 0;

producer() {
    while (true) {
        item = produceItem();
        P(itemsForForConsumer);
        P(spacesForProducer);
        P(access);
        b[in] = item;
        V(access);
        V(access);
        V(itemsForForConsumer);
        in = (in + 1) % SIZE;
    }
}

consumer() {
    while (true) {
        item = b[out];
        P(access);
        V(access);
        V(spacesForProducer);
        out = (out + 1) % SIZE;
        consumeItem(item);
    }
}
Semaphore-Based Solution

Semaphore access = 1;
Semaphore spacesForProducer = SIZE;
Semaphore itemsForConsumer = 0;

producer() {
    while (true) {
        item = produceItem();
P(spacesForProducer);
P(access);
P(access);
b[in] = item;
V(access);
V(access);
V(itemsForConsumer);
in = (in + 1) % SIZE;
    }
}

consumer() {
    while (true) {
        item = b[out];
P(itemsForConsumer);
P(access);
P(access);
V(access);
V(access);
V(spacesForProducer);
out = (out + 1) % SIZE;
consumeItem(item);
    }
}

for synchro
Notes About the Solution

- One semaphore for mutual exclusion
- Two semaphores for synchronization

- You should consider what happens if
  - swap order of $P()$’s in consumer or producer
  - swap order of $V()$’s in consumer or producer
Readers/Writers Problem

• Readers and writers have access to common file

• Criteria:
  – Any # of readers may read @ same time
  – Only one writer may write @ time
  – If any writer is writing, no reader may read
/* semaphores, vars here */

reader() {
    while (true) {
        /* protection code */
        readFile();
        /* protection code */
    }
}

writer() {
    while (true) {
        /* protection code */
        writeToFile();
        /* protection code */
    }
    }
```
read() {
    int readerCount = 0;
    Semaphore countAccess = 1;
    Semaphore fileAccess = 1;
    while (true) {
        P(countAccess);
        readerCount++;
        if (readerCount == 1)
            P(fileAccess);
        V(countAccess);
        readFile();
        P(countAccess);
        readerCount--;
        if (readerCount == 0)
            V(fileAccess);
        V(countAccess);
    }
}

writer() {
    while (true) {
        P(fileAccess);
        writeToFile();
        V(fileAccess);
        V(countAccess);
    }
}
```
Readers/Writers Solution

```c
int readerCount = 0;
Semaphore countAccess = 1;
Semaphore fileAccess = 1;

reader() {
    while (true) {
        P(countAccess);
        readerCount++;
        if (readerCount == 1)
            P(fileAccess);
        V(countAccess);
        readFile();
        P(countAccess);
        readerCount--;
        if (readerCount == 0)
            V(fileAccess);
        V(countAccess);
    }
}

writer() {
    while (true) {
        P(fileAccess);
        writeToFile();
        V(fileAccess);
    }
}
```

shared variable, so mutex
Readers/Writers Solution

```c
reader() {
    int readerCount = 0;
    Semaphore countAccess = 1;
    Semaphore fileAccess = 1;

    while (true) {
        P(countAccess);
        readerCount++;
        if (readerCount == 1) {
            P(fileAccess);
            V(countAccess);
        }
        readFile();
        V(countAccess);
    }
}

writer() {
    while (true) {
        P(fileAccess);
        writerToFile();
        V(fileAccess);
        P(countAccess);
        readerCount--;
        if (readerCount == 0) {
            V(fileAccess);
            V(countAccess);
        }
    }
}
```

if the only reader, make sure no writer present
Readers/Writers Solution

```c
reader() {
    int readerCount = 0;
    Semaphore countAccess = 1;
    Semaphore fileAccess = 1;
    readerCount++;
    if (readerCount == 1)
        P(fileAccess);
    V(countAccess);
    readFile();
    P(countAccess);
    readerCount--;
    if (readerCount == 0)
        V(fileAccess);
    V(countAccess);
}

writer() {
    while (true) {
        P(fileAccess);
        writeToFile();
        V(fileAccess);
        V(countAccess);
    }
}
```

if last reader, allow writers to go
Readers/Writers Solution

```c
reader() {
    int readerCount = 0;
    Semaphore countAccess = 1;
    Semaphore fileAccess = 1;

    while (true) {
        readerCount++;
        if (readerCount == 1)
            P(fileAccess);
        P(countAccess);
        if (readerCount == 0)
            V(fileAccess);
        V(countAccess);
        readFile();
    }
}

writer() {
    while (true) {
        P(fileAccess);
        writeToFile();
        P(countAccess);
        readerCount--;
        if (readerCount == 0)
            V(fileAccess);
        V(fileAccess);
    }
}
```
A Different IPC Mechanism

- **Monitor**: special software module
  - one or more methods
  - local data (variables, data structures)
Monitors

- **Conditions:**
  - local data accessed only by monitor’s methods
  - process enters monitor only via method call
  - only one process executing in monitor at a time

- Monitors are programming language construct

- Mutex provided automagically
  - i.e., *guarantees* mutex
    
  compiler implements; usually w/ semaphore

- Synchro not automatic
  - need a mechanism...
Condition Variables

• Monitors provide mutex automatically
  – but once inside, what if need to block?
  – e.g., producer finds full buffer

• *Condition variable*: allows a process to block itself

• *wait(c)*: suspend execution on condition variable c
  – monitor then available for another process

• *signal(c)*: resume execution of some process on c
  – if many, choose one; if none, ignore

• If process in monitor signals and none waiting, signal is lost!
  – different than semaphores
Signal Semantics

• Options:
  – signaling process continues in monitor, signaled process to ready state (Mesa)
    • see caveat in Nachos comments
  
  – signaling process immediately suspends, signaled process executes (Hoare)
  
  – signaling process exits monitor immediately (Brinch Hansen)
    • *signal* must be final statement in monitor procedure
Monitor/Semaphore Recap

• Monitors:
  – programming language construct
  – guarantee mutual exclusion
    • makes parallel programming much less error prone
  – do not provide automatic synchro
    • still possible to make mistakes
    • at least confined to monitor

• Semaphores:
  – low-level implementation (probably with TSL)
  – roll-your-own mutex & synchro
monitor PC {
    condition full, empty; // condition variables
    int count; // items in buffer
    someType buffer[]; // common array
    int in, out; // indices for prod/cons

    insert(someType item) {
        if (count == N)
            wait(full); // block on CV
        buffer[in++] = item; // details omitted
        count++; // wake, if any
    }

    someType remove() {
        if (count == 0)
            wait(empty);
        item = buffer[out++];
        count--;
        if (count == N-1)
            signal(full);
        return item;
    }
}
PC pcMon = new PC(); // instance of monitor

producer() {
    while (true) {
        item = produceItem();
        pcMon.insert(item);
    }
}

consumer() {
    while (true) {
        item = pcMon.remove();
        consumeItem(item);
    }
}

Note: mutex via monitor construct;
synchro via your method implementations (previous slide)
Monitors in Practice

- C/C++ do not provide monitor support

- Java does provide monitor support
  - no explicit condition variables
  - uses methods \texttt{wait} and \texttt{sleep} instead
And in Nachos?

- Nachos provides no monitor construct

- Same effect achieved via *locks* & *condition variables*
  
  - acquiring lock : entering “monitor”
  - wait on condition var : block w/in “monitor”

- must have lock to use condition variable
  - signal semantics: Mesa-style
Nachos 1 #2,4: Synchro Problems

- **Where do I implement?**
  - look at `threadtest.cc`
  - copy to new files containing solutions for 2, 3, 4
  - you can use *some* global vars in those implementations

- **Where do I invoke the solution?**
  - Ans: Where is(are) the `threadtest.cc` function(s) invoked?

- **How do I get my solutions to compile?**
  - modify `Makefile.common` in top level directory:
    - add to `THREAD_H` any header files you create
    - add to `THREAD_C` any source files you create
    - add to `THREAD_O` the associated object file name
  - then, execute `make` in top level directory
    (`Makefile` in `threads/` directory will be updated for you)