CS 360 Internet Programming

Concurrent Programming

*Shared Memory Synchronization with Monitors, Message Passing Synchronization, and Deadlock*

Daniel Zappala
Computer Science
Brigham Young University
1. Shared Memory Synchronization with Monitors
   - Definitions
   - Hoare Monitor
   - Lampson and Redell Monitor
   - Monitors in Non-Supporting Languages

2. Message Passing
   - Definitions
   - Mailbox
   - Direct Communication
   - UNIX Sockets

3. Deadlock
   - Definitions
   - Deadlock Example
   - Preventing Deadlock
Monitor

difficult to get semaphores right
  - match wait and signal
  - put in right order
  - scattered throughout code

monitor: programming language construct
  - equivalent functionality
  - easier to control
  - mutual exclusion constraints can be checked by the compiler
  - used in versions of Pascal, Modula, Mesa
  - Java also has a Monitor object but compliance cannot be checked at compile time
Shared Memory Synchronization with Monitors
Message Passing
Deadlock

Hoare Monitor

- monitor can only be entered through procedures
- data can only be accessed by procedures
- only one process or thread in monitor at any time
- may suspend and wait on a condition variable
- like object-oriented programming with mutual exclusion added in
Hoare Synchronization

- \textbf{cwait}(c): suspend on condition c
- \textbf{csignal}(c): wake up one thread waiting for condition c
  - do nothing if no threads waiting (signal is lost)
  - different from semaphore (number of signals represented in semaphore value)
Producer Consumer with a Hoare Monitor

```c
char buffer[BUFSIZE];
int in = 0, out = 0, count = 0;
condition notfull, notempty;

append:
1 if count == N
2    cwait(notfull);
3    buffer[in] = x;
4    in = (in + 1) % BUFSIZE;
5    count++;
6    csignal(notempty);

take:
1 if count == 0
2    cwait(notempty);
3    x = buffer[out];
4    out = (out + 1) % BUFSIZE;
5    count--;
6    csignal(notfull);
```
Producer Consumer with a Hoare Monitor

**producer:**
```c
1 char x;
2 while (True) {
3   x = produce();
4   append(x);
5 }
```

**consume:**
```c
1 char x;
2 while (True) {
3   x = take();
4   consume(x);
5 }
```

- advantages
  - moves all synchronization code into the monitor
  - monitor handles mutual exclusion
  - programmer handles synchronization (buffer full or empty)
  - synchronization is confined to monitor, so it is easier to check for correctness
  - write a correct monitor, any thread can use it (semaphores need to be placed properly in all threads)
Lampson and Redell Monitor

- Hoare monitor requires that signaled thread must run immediately
  - thread that calls `csignal()` must exit the monitor or be suspended
  - for example, when `notempty` condition signaled, thread waiting must be activated immediately or else the condition may no longer be true when it is activated
  - usually restrict `csignal()` to be the last instruction in a method (Concurrent Pascal)

- Lampson and Redell
  - replace `csignal()` with `cnotify()`
  - `cnotify(x)` signals the condition variable, but thread may continue
  - thread at head of condition queue will run at some future time
  - must recheck the condition!
  - used in Mesa, Modula-3
Producer Consumer with a Lampson Redell Monitor

```c
char buffer[BUFSIZE];
int in = 0, out = 0, count = 0;
condition notfull, notempty;

append:
1 while (count == N)
2   cwait(notfull);
3 buffer[in] = x;
4 in = (in + 1) % BUFSIZE;
5 count++;
6 cnotify(notempty);

take:
1 while (count == 0)
2   cwait(notempty);
3 x = buffer[out];
4 out = (out + 1) % BUFSIZE;
5 count--;
6 cnotify(notfull);
```
Lampson Redell Advantages

- allows processes in waiting queue to awaken periodically and reenter monitor, recheck condition
  - prevents starvation
- can also add `cbroadcast(x)`: wake up all processes waiting for condition
  - for example, append variable block of data, consumer consumes variable amount
  - for example, memory manager that frees $k$ bytes, wake all to see who can go with $k$ more bytes
- less prone to error
  - process always checks condition before doing work
What Can You Do?

- emulate a Lampson Redell Monitor with mutex and condition variables or semaphores
  - create a class with private data only
  - use the same mutex or semaphore to protect all class methods
  - use condition variables or semaphores to replace `cwait()` and `cnotify()`
  - use while loops to recheck conditions
- take your semaphores and move them inside the method call instead of outside of it (see circular buffer implementation)
Message Passing

- Needed for distributed systems: no shared memory
- Two primitives
  - `send(destination, message)`
  - `recv(source, message)`
- Blocking
  - Operating System may block sender or receiver unless you use non-blocking calls
  - When sender returns, this does not mean the message has been delivered, just accepted by the transport protocol
Message Passing with a Mailbox

- create a mailbox abstraction (managed by a separate thread)
  - threads send messages to the central mailbox
  - address message directly to another thread
  - threads poll mailbox to get messages

- provide mutual exclusion
  - use a null message as a token
  - process that gets the token can enter the critical section
  - if token is not available, block until you get it
Producer Consumer with a Mailbox

```
1 int null = 0;
2 create_mailbox(mayproduce);
3 create_mailbox(mayconsume);
4 for (int i = 1; i <= CAPACITY; i++)
5   send(mayproduce, null);

producer:
1 message pmsg;
2 while (True) {
3   receive(mayproduce, pmsg);
4   pmsg = produce();
5   send(mayconsume, pmsg);
6 }

consumer:
1 while (true) {
2   receive(mayconsume, cmsg);
3   consume(cmsg);
4   send(mayproduce, null);
5 }
```
Direct Communication

- have a thread in charge of each shared data structure or file
- web server example
  - send a message to request a new connection
  - send a message to log statistics
  - send a message to log a request
Unix Domain Socket Address Structure

```c
struct sockaddr_un {
    sa_family_t sun_family; // AF_LOCAL
    char sun_path[108]; // path name
}
```

- path name must be null terminated
UNIX Domain Server

```c
struct sockaddr_un server;
char *filename = "/tmp/mysocket";
bzero(&server, sizeof(server));
server.sin_family = AF_UNIX;
strcpy(server.sun_path, filename, sizeof(server.sun_path) - 1);
s = socket(PF_UNIX, SOCK_STREAM, 0)
if (!s) {
    perror("socket");
    exit(0);
}
if (bind(s, &server, sizeof(server)) < 0) {
    perror("bind");
    exit(0);
}
```

*call `unlink(filename)` when finished with socket*
UNIX Domain Client

1 struct sockaddr_un server;
2 char *filename = "/tmp/mysocket";
3 bzero(&server, sizeof(server));
4 server.sin_family = AF_UNIX;
5 strncpy(server.sun_path, filename, sizeof(server.sun_path) - 1);
6
7 s = socket(PF_UNIX, SOCK_STREAM, 0)
8 if (!s) {
9     perror("socket");
10     exit();
11 }
12 if (connect(s, &server, sizeof(server)) < 0) {
13     perror("connect");
14     exit();
15 }
Deadlock Definition and Conditions

- permanent blocking of a set of processes or threads that either compete for system resources or communicate with each other
- conditions
  1. mutual exclusion: only one thread may use a resource at a time
  2. hold-and-wait: a thread keeps one resource while waiting for another
  3. no preemption: a thread can’t be forced to release a resource
  4. circular wait: a cycle of threads waiting for each other
- if first three conditions hold, then deadlock is possible if circular wait occurs
- depends on execution order!
Example

1. Thread P
2. ...
3. Get A
4. ...
5. Get B
6. ...
7. Release A
8. ...
9. Release B

1. Thread Q
2. ...
3. Get B
4. ...
5. Get A
6. ...
7. Release B
8. ...
9. Release A

is deadlock possible?
Deadlock Possibilities

- Release A
- Release B
- Get A
- Get B

A Required

B Required

Progress of Q

1. Get A
2. Get B
3. Release A
4. Release B

P and Q want A
P and Q want B

deadlock inevitable

Progress of P
Revised Sample Code

1 Thread P
2 ...
3 Get A
4 ...
5 Release A
6 ...
7 Get B
8 ...
9 Release B

1 Thread Q
2 ...
3 Get B
4 ...
5 Get A
6 ...
7 Release B
8 ...
9 Release A

is deadlock possible?
Deadlock Avoided

A Required

B Required

Release A

Get A

Release B

Get B

Progress of Q

Progress of P

1

2

3

4

5

6

A Required

B Required

P and Q want A

P and Q want B
Simple Deadlock Prevention

- prevent one of the conditions from happening
- simplest to prevent is **hold-and-wait**: hold only one resource at a time
- can also prevent **circular wait**: impose ordering on resources