Operating Systems

Process Synchronization (Ch 6.1 - 6.7)

Cooperating Processes

- Consider: print spooler
  - Enter file name in spool queue
  - Printer daemon checks queue and prints

Too Much Pizza

<table>
<thead>
<tr>
<th>Time</th>
<th>Person A</th>
<th>Person B</th>
</tr>
</thead>
<tbody>
<tr>
<td>3:00</td>
<td>Look in fridge. Pizza!</td>
<td>Look in fridge. Pizza!</td>
</tr>
<tr>
<td>3:05</td>
<td>Leave for store.</td>
<td>Leave for store.</td>
</tr>
<tr>
<td>3:10</td>
<td>Arrive at store.</td>
<td>Leave for store.</td>
</tr>
<tr>
<td>3:15</td>
<td>Buy pizza.</td>
<td>Arrive at store.</td>
</tr>
<tr>
<td>3:30</td>
<td></td>
<td>Put pizza away.</td>
</tr>
<tr>
<td></td>
<td>Oh no!</td>
<td></td>
</tr>
</tbody>
</table>

“Race conditions” (ugh!)
(Hey, you! Show demo!)

Producer Consumer

- Model for cooperating processes
- Producer “produces” and item that consumer “consumes”
- Bounded buffer (shared memory)

```c
item buffer[MAX]; /* queue */
int counter; /* num items */
```

Producer

```c
item i; /* item produced */
int in; /* put next item */
while (1) {
  produce an item
  while (counter == MAX) {/*no-op*/}
  buffer[in] = item;
  in = (in + 1) % MAX;
  counter = counter + 1;
}
```

Consumer

```c
item i; /* item consumed */
int out; /* take next item */
while (1) {
  while (counter == 0) {/*no-op*/}
  item = buffer[out];
  out = (out + 1) % MAX;
  counter = counter - 1;
  consume the item
}
```
Trouble!

\[ P: \ R1 = \text{counter} \ \ (R1 = 5) \]
\[ P: \ R1 = R1 + 1 \ \ (R1 = 6) \]
\[ C: \ R2 = \text{counter} \ \ (R2 = 5) \]
\[ C: \ R2 = R2 -1 \ \ (R2 = 4) \]
\[ P: \ \text{counter} = R1 \ \ (\text{counter} = 6) \]
\[ C: \ \text{counter} = R2 \ \ (\text{counter} = 4) \]

First Try: Strict Alternation

```
int turn; /* shared, i or j */
while(1) {
    while (turn <> i) { /* no-op */}
    /* critical section */
    turn = j
    /* remainder section */
}
```

Second Try

```
int flag[1]; /* boolean */
while(1) {
    flag[i] = true;
    while (flag[j]) { /* no-op */}
    /* critical section */
    flag[i] = false;
    /* remainder section */
}
```

Third Try: Peterson’s Solution

```
int flag[1]; /* boolean */
int turn;
while(1) {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn==j){ }
    /* critical section */
    flag[i] = false;
    /* remainder section */
}
```

Multiple-Processes

```
“Bakery Algorithm”

Common data structures
boolean choosing[n];
int num[n];

Ordering of processes
– If same number, can decide “winner”
```

Critical Section

- Mutual Exclusion
  – Only one process inside critical region
- Progress
  – No process outside critical region may block
    other processes wanting in
- Bounded Waiting
  – No process should have to wait forever
    (starvation)
- Note, no assumptions about speed!
Multiple-Processes

choosing[i] = true;
num[i] = max(num[0],num[1] ...) + 1
choosing[i] = false;
for (j=0; j<n; j++) {
    while(choosing[j]) { }
    while( num[j]!=0 && 
         (num[j],j)<(num[i],i) ) {} 
} /* critical section */
num[i] = 0;

Synchronization Hardware

∗ Test-and-Set: returns and modifies atomically
int Test_and_Set(int target) {
    int temp;
    temp = target;
    target = true;
    return temp;
}

Synchronization Hardware

while(1) {
    while (Test_and_Set(lock)) { }
} /* critical section */
lock = false;
/* remainder section */

Semaphores

∗ Does not require “busy waiting”
∗ Semaphore S (shared, often initially =1)
    – integer variable
    – accessed via two (indivisible) atomic operations
    wait(S): S = S - 1
        if S<0 then block(S)
    signal(S): S = S + 1
        if S<=0 then wakeup(S)

Critical Section w/Semaphores

semaphore mutex; /* shared */

while(1) { 
    wait(mutex);
} /* critical section */
signal(mutex);
/* remainder section */

Semaphore Implementation

∗ How do you make sure the signal and the wait operations are atomic?
Semaphore Implementation
- Disable interrupts
  - Why is this not evil?
  - Multi-processors?
- Use correct software solution
- Use special hardware, i.e.- Test-and-Set

Design Technique: Reducing a Problem to a Special Case
- Simple solution not adequate
  - ex: disabling interrupts
- Problem solution requires special case solution
  - ex: protecting S for semaphores
- Simple solution adequate for special case
- Other examples:
  - name servers, on-line help

Trouble!
```
signal(S)
/* cr */
wait(S)
```  
```
wait(S)
/* cr */
wait(S)
```  
```
Process A
wait(S)
wait(Q)
...
```  
```
Process B
wait(Q)
wait(S)
```  

Project 2: Mini Chat
- Shared memory
- Concurrent processes
- Semaphores

Outline
- Processes Synchronization (Ch 6.1 - 6.7)
  - Shared memory ✓
  - Hardware ✓
  - Semaphores ←
  - Classical Problems
  - Other methods
- Interprocess Communication (Ch 4.6)
- Threads (Ch 4.5)

Review
- What is “mutual exclusion violation”?
  - Why do we care?
- What is “busy waiting”?
- How does a semaphore work?
SOS Semaphore Implementation

- Semaphore structure
  - array in OS
  - integer id to use in process
- AttachSemaphore(key), returns sid
- DetachSemaphore(sid)
- SignalSemaphore(sid)
- WaitSemaphore(sid)

Classical Synchronization Problems

- Bounded Buffer
- Readers Writers
- Dining Philosophers

Dining Philosophers

- Philosophers
  - Think
  - Sit
  - Eat
  - Think
- Need 2 chopsticks to eat

Dining Philosophers

Philosopher i:
while (1) {
  /* think... */
  wait(chopstick[i]);
  wait(chopstick[i+1 % 5]);
  /* eat */
  signal(chopstick[i]);
  signal(chopstick[i+1 % 5]);
}

Other Solutions

- Allow at most N-1 to sit at a time
- Allow to pick up chopsticks only if both are available
- Asymmetric solution (odd L-R, even R-L)
Readers-Writers

* Readers only read the content of object
* Writers read and write the object
* Critical region:
  - No processes
  - One or more readers (no writers)
  - One writer (nothing else)
* Solutions favor Reader or Writer

Readers-Writers

Reader:
wait(mutex);
readcount = readcount + 1;
if (readcount==1) wait(wrt);
signal(mutex);
/* read stuff */
wait(mutex);
readcount = readcount - 1;
if (readcount==0) signal(wrt);
signal(mutex);

Readers-Writers

Shared:
semaphore mutex, wrt;
int readcount;

Writer:
wait(wrt)
/* write stuff */
signal(wrt);

"Critical Region"

High-level construct
region X do S
X is shared variable
S is sequence of statements

Compiler says:
wait(x-mutex)
S
signal(x-mutex)

"Critical Region"

Deadlocks still possible:
- Process A:
  region X do
  region Y do S1;
- Process B:
  region Y do
  region X do S2;

Conditional Critical Regions

High-level construct
region X when B do S
X is shared variable
B is boolean expression (based on c.r.)
S is sequence of statements
Bounded Buffer

Shared:
struct record {
    item pool[MAX];
    int count, in, out;
};
struct record buffer;

Bounded Buffer Producer

region buffer when (count < MAX){
    pool[in] = i; /* next item*/
    in = in + 1;
    count = count + 1;
}

Bounded Buffer Consumer

region buffer when (count > 0){
    nextc = pool[out];
    out = (out + 1) % n;
    count = count - 1;
}

Monitors

✦ High-level construct
✦ Collection of:
    – variables
    – data structures
    – functions
    – Like C++ class
✦ One process active inside
✦ “Condition” variable
    – not counters like semaphores

Monitor Producer-Consumer

monitor ProducerConsumer {
    condition full, empty; /* not semaphores */
    integer count;
    /* function prototypes */
    void producer();
    void consumer();
    void enter(item i);
    item remove();
}

Monitor Producer-Consumer

void producer() {
    item i;
    while (1) {
        /* produce item i */
        ProducerConsumer.enter(i);
    }
}
void consumer() {
    item i;
    while (1) {
        i = ProducerConsumer.remove();
        /* consume item i */
    }
}
Monitor Producer-Consumer

```c
void enter (item i) {
    if (count == N) wait(full);
    /* add item i */
    count = count + 1;
    if (count == 1) then signal(empty);
}

item remove () {
    if (count == 0) then wait(empty);
    /* remove item into i */
    count = count - 1;
    if (count == N-1) then signal(full);
    return i;
}
```

Other IPC Synchronization

- Sequencers
- Path Expressions
- Serializers
- ...
- All essentially equivalent in terms of semantics. Can build each other.

Ex: Cond. Crit. Region w/Sem

```c
region X when B do S {
    wait(x-mutex);
    if (!B) {
        x-count = x-count + 1;
        signal(x-mutex);
        wait(x-delay);
        /* wakeup loop */
        x-count = x-count -1
    }
    /* remainder */
```

Ex: Wakeup Loop

```c
while (!B) {
    x-temp = x-temp + 1;
    if (x-temp < x-count)
        signal(x-delay);
    else
        signal(x-mutex);
    wait(x-delay);
}
```

Ex: Remainder

```c
S;
if (x-count > 0) {
    x-temp = 0;
    signal(x-delay);
} else
    signal(x-mutex);
```

Trouble?

- Monitors and Regions attractive, but ...
  - Not supported by C, C++, Pascal ...
  - Semaphores easy to add
- Monitors, Semaphores, Regions ...
  - Require shared memory
  - Break on multiple CPU (w/own mem)
  - Break distributed systems
- Message Passing!
Message Passing

- Communicate information from one process to another via primitives:
  - send(dest, &message)
  - receive(source, &message)
- Receiver can specify ANY
- Receiver can block (or not)

Producer-Consumer

```c
void Producer() {
    while (TRUE) {
        /* produce item */
        build_message(&m, item);
        send(consumer, &m);
        receive(consumer, &m); /* wait for ack */
    }
}

void Consumer {
    while(1) {
        receive(producer, &m);
        extract_item(&m, &item);
        send(producer, &m); /* ack */
        /* consume item */
    }
}
```

Consumer Mailbox

```c
void Consumer {
    for (i=0; i<N; i++)
        send(producer, &m); /* N empties */
    while(1) {
        receive(producer, &m);
        extract_item(&m, &item);
        send(producer, &m); /* ack */
        /* consume item */
    }
}
```

New Troubles with Messages?

- Scrambled messages (checksum)
- Lost messages (acknowledgements)
- Lost acknowledgements (sequence no.)
- Process unreachable (down, terminates)
- Naming
- Authentication
- Performance (from copying, message building)