Operating System I

Process Synchronization

Cooperating Processes

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

Producer Consumer

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

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

P: R1 = counter \( (R1 = 5) \)
P: R1 = R1 + 1 \( (R1 = 6) \)
C: R2 = counter \( (R2 = 5) \)
C: R2 = R2 - 1 \( (R2 = 4) \)
P: counter = R1 \( (counter = 6) \)
C: counter = R2 \( (counter \) 

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!

First Try: Strict Alternation

```c
int turn; /* shared, i or j */

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

Second Try

```c
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

```c
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
  ```c
  boolean choosing[n];
  int num[n];
  ```
- Ordering of processes
  - If same number, can decide “winner”
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 */
um[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)) { }
  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

Classical Synchronization Problems

- Bounded Buffer
- Readers Writers
- Dining Philosophers

Dining Philosophers

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

Trouble!

<table>
<thead>
<tr>
<th>Process A</th>
<th>Process B</th>
</tr>
</thead>
<tbody>
<tr>
<td>wait(S)</td>
<td>wait(Q)</td>
</tr>
<tr>
<td>wait(Q)</td>
<td>wait(S)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

/* cr */

Dining Philosophers

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

Other Solutions?
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

**Shared:**

semaphore mutex, wrt;
int readcount;

**Writer:**

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

**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);

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;
    ... wait(x-mutex)
  
  - Process B:
    
    region Y do
    region X do S2;
    ... wait(y-mutex)

Process A

- wait(x-mutex)
- ... wait(y-mutex)

Process B

- wait(y-mutex)
- ... wait(x-mutex)
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:
```c
struct record {
    item pool[MAX];
    int count, in, out;
};
struct record buffer;
```

Bounded Buffer Producer

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

Bounded Buffer Consumer

```c
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

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

```c
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?

New Troubles

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