CS571: Operating Systems

CLASS 4 : 12 FEB 2003
16:30 - 19:10

Who Uses What

- OS creates/controls resources in system
- allows tasks to access resources
- some can be shared, e.g., memory
- some cannot be shared, e.g., a printer

Too Many Cooks

- many resources under OS control cannot be shared
- OS needs way to enforce strict one-at-a-time usage
- not as easy as it sounds

Race Conditions

- problem arises due to race conditions:
  - the outcome of running a set of processes varies depending on the order in which events occur
  - SG&G has example on page 173
- we look at a different example here...

Example Race Condition

| slots in spool directory (acts like a queue): contain names of files to be printed |
|----------------------------------|-------------------------------|
| 4: abc.txt                       | 5: file1.ps                    |
| 6: file2.ps                      | 7: in = 7                      |
| slot to print next               | slot to put filename into next |

process A wants to add name to spool dir:
next_free_slot = in++; 
spool[next_free_slot] = "A_s_file2print"
**Example Race Condition**

process A wants to add name to spool dir:

\[
\text{next_free_slot} = \text{in}++; \\
\text{spool}[\text{next_free_slot}] = "A_s_file2print"
\]

out = 4

STOP

process B wants to add name to spool dir:

\[
\text{next_free_slot} = \text{in}++; \\
\text{spool}[\text{next_free_slot}] = "B_s_file2print"
\]

out = 4

STOP

**Example Race Condition**

process A wants to add name to spool dir:

\[
\text{next_free_slot} = \text{in}++; \\
\text{spool}[\text{next_free_slot}] = "A_s_file2print"
\]

in = 7

STOP

process B wants to add name to spool dir:

\[
\text{next_free_slot} = \text{in}++; \\
\text{spool}[\text{next_free_slot}] = "B_s_file2print"
\]

in = 7

STOP

**Race to the Printer**

- which file is printed (A's or B's) depends on when the timeslice occurs
- race conditions are very hard to debug
- spooler example:
  - nearly always works correctly
  - spool directory indexes (in/out) always consistent
  - on rare occasions, loses a job

**Conditions for Race Conditions**

1. access to a common object:
   - 1 or more processes have access to some object that is commonly accessible
   - e.g., print spool directory

2. asynchronous behaviour:
   - don’t know when some process will access the common object
   - don’t know when you (process) will be interrupted (e.g., timeslice)
Critical Sections

- **Critical section**:
  - section of code that must not be simultaneously accessed by more than one process
  - code = instructions + variables
  - otherwise unpredictable behaviour (race condition) can result
  - how to assure one-at-a-time use of a critical section?

****

Mutex Requirements

1. mutual exclusion:
   - must guarantee only 1 process runs in the CS at a time

2. no assumptions made about
   1. relative process speeds
   2. number of processors

3. a process waiting outside the CS must not interfere with other processes

4. no process requiring access to the CS can be delayed indefinitely from entering: no deadlock or starvation
Mutex Requirements

4. no process requiring access to the CS can be delayed indefinitely from entering: no deadlock or starvation.

Solving the Mutex Problem

- solve in software:
  - introduce ‘gateway’ software: all processes wanting to enter the CS must enter via this gateway.
  - may have ‘exit portal’ software: processes leaving the CS exit here (releasing the CS for another process to use).

Mutex Software Solution 1

```c
int turn = 0;
while (turn != 0) {
    i = i; /* do nothing */
} turn = 1;
```

Mutex Software Solution 3

```c
int flag[n]; /* n is # proc involved in mutex */
while (flag[0]) {
    j = j; /* do nothing */
} flag[0] = 1;
```
Mutex Software Solution 4

```c
int flag[n]; /* n is # procs involved in mutex */
/* process 0 */
flag[0] = 0;
while (flag[1]) {
  flag[0] = 1;
  delay a short while
  flag[0] = 1;
}
critical section
flag[0] = 0;
flag[1] = 0;
```

Mutex Software Solution 5

```c
int flag[n]; /* n is # procs involved in mutex */
/* process 0 */
int turn;
/* process 1 */

int flag[0];
int flag[1];
flag[0] = 1;
while (flag[1]) {
  if (turn == 1) {
    flag[0] = 0;
    while (turn) do nop;
    flag[0] = 1;
  }
}
critical section
flag[0] = 1;
critical section
flag[0] = 0;

int turn = 1;
flag[0] = 0;
flag[1] = 1;
while (flag[1]) {
  if (turn != 1) {
    flag[1] = 0;
    while (!turn) do nop;
    flag[1] = 1;
  }
}
critical section
flag[1] = 0;
critical section
flag[1] = 0;
```

And the winner is...

- Dekker’s solution, 1965 (previous slide)
- A later (1981), simplified version:

```
/* process 0 */
flag[0] = 1;
Turn = 0;
while ((Turn == 0) && (flag[1])) nop;
critical section
flag[0] = 0;
```

```
/* process 1 */
flag[1] = 1;
Turn = 1;
while ((Turn == 1) && (flag[0])) nop;
critical section
flag[1] = 0;
```

Peterson’s Solution

- Simpler
- Generalizes to n processes
- Can be packaged as pair of routines:
  - enterCS(): called by code that wants to enter CS
    - proceeds directly if CS available
    - blocks otherwise until CS is available
  - leaveCS(): called when code exits the CS
    - releasing CS for any others to use
- In SC&G as “Algorithm 3”, figure 7.6

Mutex and Hardware

- Can mutex be implemented directly in hardware?
Mutex and Hardware

- can mutex be implemented directly in hardware?
- could mask all interrupts while in CS
  - no time slice context switches, so process in CS guaranteed to use until done
  - problems?

 Mutex and Hardware

- can mutex be implemented directly in hardware?
- could mask all interrupts while in CS
  - no time slice context switches, so process in CS guaranteed to use until done
  - problems?
  - if process enters loop inside CS
  - miss clock, other important event interrupts

 Mutex and Hardware

- hardware help exists:
  1. "test & set"
     - atomic operation: cannot be interrupted once begun
     - tests a memory location’s value and changes it
     - test value of argument
       - if 0 replace by 1 and return true
       - else don’t change and return false

 Mutex and Hardware

- hardware help exists:
  1. "test & set"
  2. "exchange":
     - atomic operation: cannot be interrupted once begun
     - exchanges the contents of two locations ($\geq 1$ is a register)
     - if one is memory location, access to that location is blocked (to all) during operation
     - supported on Pentium (xchg)

 Test and Set: Pro

- advantages:
  - simple, dependable
  - applicable to any number of processes on any number of CPUs sharing memory
  - can use to support multiple CS, each with its own variable
Test and Set: Pro and Con

**advantages:**
- simple, dependable
- applicable to any number of processes on any number of CPUs sharing memory
- can use to support multiple CS, each with its own variable

**disadvantages:**
- can waste time in spin loop
- can have starvation
- can have deadlock

What Can Programmers Use?

a basic roadway intersection

how are these critical sections controlled on real roads?

Semaphores

- a semaphore
  - has an integer count variable
  - has a queue to hold waiting processes
- 3 operations on semaphores...

Operations on Semaphores

1. wait(s)
   - decrement semaphore s.count
   - if s.count < 0 then block invoking process (put on queue of semaphore s)

2. signal(s)
   - increment semaphore s.count
   - if s.count ≤ 0 unblock a process in the queue of semaphore s
Operations on Semaphores

1. wait(s)
   - decrement semaphore s
   - if s ≤ 0 then block invoking process (put on queue of semaphore s)
2. signal(s)
   - increment semaphore s
   - if s < 0 unblock a process in the queue of semaphore s
3. initialize s.count to a value > 0

Semaphore Terminology

- strong semaphore: remove from queue using FIFO
- weak semaphore: no statement of how to dequeue
- binary semaphore: count may only be 1 or 0
- counting semaphore: count may take on any value
- wait(s) sometimes written P(s)
- signal(s) sometimes written as V(s)

wait(s), signal(s) implementation note

- what must be true in order for wait() and signal() to work properly?
  - operation must be atomic
    - can’t have > 1 process performing a wait() or signal() on the same semaphore at the same time
- critical section problem; usually solve using
  - system call
  - makes use of hardware mechanism (ts, xchg, …)
  - often just mask interrupts (not the best plan)

Using semaphores

```c
/* process 0 */
wait(mutex);
signal(mutex);
/* process 1 */
wait(mutex);
signal(mutex);
```
Using semaphores

- why initialize counter to 1 in this example?

Semaphores in Real Problems

- producer/consumer problem
  - shared memory buffer: finite size
  - usual implementation: ring buffer

Producer/Consumer Problem

- solve with semaphores
- is there a critical section?

```c
producer
    sh_buf[i++] % n = stuff;
consumer
    thing = sh_buf[j++] % n;
```

- producer must not write into already full buffer
- consumer must not consume what hasn’t yet been produced
- a.k.a. bounded buffer problem

- 1 critical section:
  - access to same buffer element (race condition)
  - any other resources to protect with semaphores?
  - anything to which access may need to be synchronized?
Producet/Consumer Problem

• yes:
  • producer needs "emptiness" as resource, must block when none left
  • consumer needs “fullness” as resource, must block when none left

Semaphore Solution to Producer Consumer

```c
producer()
{
    int item;
    while(1) {
        produce_item(&item);
        wait(empty);
        wait(mutex);
        enter_item(&item);
        signal(mutex);
        signal(full);
    }
}
```

```c
customer()
{
    int item;
    while(1) {
        wait(full);
        wait(mutex);
        remove_item(&item);
        signal(mutex);
        signal(empty);
        consume_item(&item);
    }
}
```

Semaphore mutex = 1;
Semaphore full  = 0;
Semaphore empty = N;

Ideal Solution?

• this solution does work
• semaphores must be used with caution...

Semaphore Solution to Producer Consumer

```c
producer()
{
    int item;
    while(1) {
        produce_item(&item);
        wait(mutex);
        wait(empty);
        enter_item(&item);
        signal(mutex);
        signal(full);
    }
}
```

```c
customer()
{
    int item;
    while(1) {
        wait(full);
        wait(mutex);
        remove_item(&item);
        signal(mutex);
        signal(empty);
        consume_item(&item);
    }
}
```

Semaphore mutex = 1;
Semaphore full  = 0;
Semaphore empty = N;

Ideal Solution?

• careless use of semaphores can lead to
  • deadlock
  • starvation if dequeuing method allows a queued process to remain queued indefinitely (e.g., if used LIFO instead of FIFO)

• other common synchronization problems:
  • readers & writers: see SG&G section 7.6.2
  • Dining Philosophers: see SG&G section 7.6.3

Monitors

• a high-level language construct
• a software module containing
  • ≥ 1 procedures (callable from outside monitor)
  • an initialization sequence
  • local data (not visible outside monitor)
Monitors

- a high-level language construct
- a software module containing
- main features:
  - process enters monitor by invoking one of the procedures inside the monitor
  - at most 1 process can execute in monitor at a time
  - any other process attempting access is blocked \(\Rightarrow\) mutual exclusion is provided

Monitors

- what happens if:
  - process inside monitor becomes suspended (e.g., waiting for something)?

Monitors

- what happens if:
  - process inside monitor becomes suspended (e.g., waiting for something)?
  - would like to:
    - suspend that process
    - release monitor so others can use it
    - be able to ‘wake’ the suspended process up later to it can resume and finish

Monitors

- monitors provide condition variables with 2 defined operations:
  - cwait\(c\) suspend invoking process on condition \(c\); monitor becomes available to other processes
  - csignal\(c\) resume some process now waiting on condition \(c\); if none, do nothing (signal is lost)

Using monitores

```
monitor prodcon {
  char buffer[N];
  int maxin, nextout;
  int count;
  int notempty, notfull;

  void append(char x) {
    if (count == N) 
      cwait(notfull);
    buffer[nextin] = x;
    nextin = (++nextin) % N;
    count++;
    csignal(notempty);
  }

  void take(char x) {
    if (count == 0) 
      cwait(notempty);
    x = buffer[nextout];
    nextout = (++nextout) % N;
    --count;
    csignal(notfull);
  }

  nextin = nextout = count = 0;
}
```
Using monitors

```c
monitor
void producer()
    char x;
    while(1) {
        produce(&x);
        append(x);
    }
void consumer()
    char y;
    while(1) {
        take(y);
        consume(&y);
    }
```

Waking up in a Monitor

- what happens when a process is woken up on a `csignal(c)`?
- remember: the procedure issuing the `csignal()` must now be running in the monitor...

Waking up in a Monitor

- so process issuing `csignal()` and woken up process now both run?
- not permitted in a monitor

Different Strategies for wake-ups

1. Hoare's model requires that process issuing `csignal()` must
   - immediately exit monitor
   - i.e., `csignal()` is last executed statement in that process
   - be suspended
   - use 'urgent' queue in monitor for this process, rather than make it wait a full rotation through input queue

Different Strategies for wake-ups

2. Lampson & Redell replace `csignal(c)` with `cnotify(c)`:
   - wait queue `c` is notified of a wake-up: process at head of that queue is 'marked' for resumption
   - notifying process continues running
   - marked process resumes 'later' when monitor becomes available

Different Strategies for wake-ups

- marked process resumes 'later' when monitor becomes available
- but another process might have entered and run in the monitor before the awoken process resumes
Different Strategies for wake-ups

- marked process resumes 'later' when monitor becomes available
- but another process might have entered and run in the monitor before the awoken process resumes
- so awoken process must check condition variable before continuing

Differences csignal() and cnotify()

- with csignal():
  ```
  void append(char x) {
  if (count == N)
       cwait(notfull);
  buffer[nextin] = x;
  nextin = ++nextin % N;
  count++;
  csignal(notempty);
  }
  ```

- with cnotify():
  ```
  void append(char x) {
  while (count == N)
       cwait(notfull);
  buffer[nextin] = x;
  nextin = ++nextin % N;
  count++;
  csignal(notempty);
  }
  ```

csignal() and cnotify()

- what's the advantage of cnotify()?  
  - less error prone: awoken process always checks its condition variable  
  - easier scheduling constraints: with csignal() is critical to ensure awoken process resumes before any other process can begin running in monitor  
  - else condition variables may change  
  - reduces process switching overhead  
  - csignal() requires at least extra switch of putting signaller to sleep and waking up signalled process  
  - extend cnotify() to cbroadcast()

Thread Synchronization in Java

- every object has one lock
- any method on an object may be declared with attribute synchronized  
  - meaning: to enter this method, lock must be obtained  
  - invoking thread becomes 'owner' of lock  
  - entry to synchronized method  
  - succeeds if lock is available  
  - fails otherwise: invoking thread blocks

Thread Synchronization in Java

- blocked waiting thread placed in entry set

- as threads stops running in this method, another waiting thread is selected from entry set
Thread Synchronization in Java

- a thread may want to wake up a waiting thread
  - if it provides event for which a thread is waiting
- issues notify() to wake up a thread in wait set
  - causes some waiting thread to move from wait set to entry set
  - thread so moved is chosen arbitrarily; thread doing notify() doesn't choose
- formerly waiting thread eventually resumes running in method
  - what must this thread do first?

- determine if event for which it waits has happened
  - waits again if not
- a thread may issue notifyAll() to move all threads from wait set to entry set
  - what does this overall mechanism resemble?

Thread Synchronization in Java

- looks a lot like a monitor except:
  - no named condition variables
- Java does not provide semaphores
  - but function can be derived using wait() and notify() with synchronized methods

And now the bad news...

- techniques described so far work when
  - 1 or more CPUs
  - all CPUs share a single memory
- none of these techniques work if the CPUs each work with their own local memory
  - no means of info exchange between machines
- so need some other technique...

Got the message?

- new method: message passing
  - exchange of messages between systems
  - 2 operations defined on messages:

Message Passing Operations

- send(to, msg)
  - may be blocking or non-blocking
- receive(from, msg)
  - may block waiting for a msg, or,
  - return immediately if no msg
3 scenarios

1. blocking send and blocking receive
   - sender and receiver run in “lock-step”
   - good for tight synchronization
   - a.k.a. rendez-vous

2. non-blocking send and blocking receive
   - sender continues after send
   - receiver blocks waiting for requested msg to arrive
     - can be source-specific
     - generally, most useful model
       - because receiver can’t proceed until msg arrives
       - e.g., client-server for file server

3. non-blocking send and non-blocking receive

Scenario 2: Some Pros and Cons

✓ good for client/server interactions
   • errors may lead to many (e.g., non-stop) send msgs
     • soaks up resources …
     • denial of service
   • programmer has to ensure a sent msg has been received

A solution…

• what happens if:
  1) sender fails to send msg?
  2) msg is lost?

  use non-blocking receive()
    • any down-side?

A solution…

• use non-blocking receive():
  • solves problem but what if receive() is executed before send() completes?
  • other variations:
    • probe for msgs waiting before actually issuing receive() request
    • allow receive() from multiple sources, any of which can unblock the receive()
To and From

- what are to and from in send(to,msg) and receive(from,msg)?

 Processes

- so how do we identify sender and recipient?
- 2 ways:

Message Addresses

- direct addressing:
  - sender uses identifier of target process as to
  - receiver:
    - may need to designate particular sender to receive
    - from so must know, a priori, who will be sending
    - may only get from value after receipt of message ... i.e., accepts messages from anyone (e.g., print server)

Indirect Addressing

- supports multiple relationships between sender & receiver
  - 1:1 private "comm link" between sender & receiver
  - many:1 std for client–server interaction (one process provides service for many clients)
  - 1:many 1 sender → many recipients = broadcast

Indirect Addressing

- supports multiple relationships between sender & receiver
  - mailboxes here referred to as ports
  - many:1 std for client–server interaction (one process provides service for many clients)
  - 1:many 1 sender → many recipients = broadcast
Processes and Mailboxes

- relationship between processes and mailboxes may be:
  - static:
    - ports bound statically to particular processes
    - 1:1 often assigned statically and permanently
  - dynamic:
    - typically when many possible senders
    - senders connect() for interaction then disconnect()

What messages look like

- have two parts:
  - header with addressing and control information
    - often fixed-length
  - data with actual message
    - usually variable length

<table>
<thead>
<tr>
<th>Field</th>
<th>Length</th>
<th>Service Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>vers</td>
<td>len</td>
<td>total length</td>
</tr>
<tr>
<td>ident</td>
<td>flags</td>
<td>fragment offset</td>
</tr>
<tr>
<td>time</td>
<td>proto</td>
<td>header checksum</td>
</tr>
<tr>
<td>source address</td>
<td>destination address</td>
<td>options</td>
</tr>
<tr>
<td>padding</td>
<td>message data</td>
<td></td>
</tr>
</tbody>
</table>

Receiving

- messages queue up at receiver
- can drain queue using:
  - FIFO (typically)
  - multiple queues for prioritized messages
  - receiver may choose according to certain criteria

Message Passing for Producer/Consumer

```c
void producer() {
    message pmsg;
    while(1) {
        receive(mayproduce, pmsg);
        pmsg = produce();
        send(mayconsume, pmsg);
    }
}

void consumer() {
    message cmsg;
    while(1) {
        receive(mayconsume, cmsg);
        consume(cmsg);
        send(mayproduce, null);
    }
}
```

Assignment 2

- now available on the course web site
- due 30 Sep 2003
- exercise in
  - use of threads
  - synchronization using semaphores
  - controlling operation of tty device
- must be done in C