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

slot to print next

out = 4

slot to put filename into next

in = 7

Example Race Condition

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

out = 4

in = 7
Example Race Condition

process A wants to add name to spool dir:

next_free_slot = in++;
spool[next_free_slot] = "A_s_file2print"

out = 4
in = 7

process B wants to add name to spool dir:

next_free_slot = in++;
spool[next_free_slot] = "B_s_file2print"

out = 4
in = 7
Example Race Condition

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

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?

- **mutual exclusion (mutex)**:  
  - only 1 process at-a-time is using the CS  
  - the means of enforcing that at most 1 process accesses a critical section at any one time

- what requirements do we have of a mutex solution?
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
Mutex Requirements

1. mutual exclusion
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

P₁ has A needs B to advance releases B and A when finished

P₂ has B needs A to advance releases A and B when finished

two (or more) processes each waiting for the other to release a resource being held so processes can advance

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

one process is consistently denied access to a resource while other processes succeed in accessing it

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)

```c
int turn = 0;

/* process 0 */
...
while (turn != 0) {
    i = i; /* do nothing */
}

turn = 1;
...

/* process 1 */
...
while (turn != 1) {
    j = j; /* do nothing */
}

critical section

turn = 0;
...
critical section
```
Mutex Software Solution 2

```c
int flag[n]; /* n is # procs involved in mutex */
/* process 0 */
... while (flag[1] == 1) {
    i = i; /* do nothing */
}
flag[0] = 1;
critical section
flag[0] = 0;
...
```

```c
/* process 1 */
... while (flag[0] == 1) {
    j = j; /* do nothing */
}
flag[1] = 1;
critical section
flag[1] = 0;
...
```

Mutex Software Solution 3

```c
int flag[n]; /* n is # procs involved in mutex */
/* process 0 */
... flag[0] = 1;
while (flag[1]) {
    i = i; /* do nothing */
}
critical section
flag[0] = 0;
```

```c
/* process 1 */
... flag[1] = 1;
while (flag[0]) {
    j = j; /* do nothing */
}
critical section
flag[1] = 0;
```
Mutex Software Solution 4

```c
int flag[n]; /* n is # procs involved in mutex */
/* process 0 */
... 
flag[0] = 1;
while (flag[1]) {
    if (turn == 1) {
        flag[0] = 0;
        while(turn) do nop;
        flag[0] = 1;
    }
}
flag[0] = 0;
```

```c
/* process 1 */
... 
flag[1] = 1;
while (flag[0]) {
    if (turn != 1) {
        flag[1] = 0;
        while(!turn) do nop;
        flag[1] = 1;
    }
}
flag[1] = 0;
```

Mutex Software Solution 5

```c
int flag[n]; /* n is # procs involved in mutex */
int turn;
/* process 0 */
... 
flag[0] = 1;
while (flag[1]) {
    if (turn == 1) {
        flag[0] = 0;
        while(turn) do nop;
        flag[0] = 1;
    }
}
turn = 1;
flag[0] = 0;
```

```c
/* process 1 */
... 
flag[1] = 1;
while (flag[0]) {
    if (turn != 1) {
        flag[1] = 0;
        while(!turn) do nop;
        flag[1] = 1;
    }
}
turn = 0;
flag[1] = 0;
```
And the winner is…

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

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

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

![Roadway Intersection Diagram]
What Can Programmers Use?

a basic roadway intersection:

critical section

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 \( s \leq 0 \) 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
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
Semaphore mutex;
mutex.count = 1;

/* process 0 */
.. 
.. 
.. 
wait(mutex);
.. 
critical section
.. 
signal(mutex);
.. 
.. 

/* process 1 */
.. 
.. 
.. 
wait(mutex);
.. 
critical section
.. 
signal(mutex);
.. 
.. 
.. 
```

© Charles M Snow 24
Using semaphores

- why initialize counter to 1 in this example?

  - 1 ‘instance of the resource’ for which we want to arrange mutual exclusion: 1 critical section
Semaphores in Real Problems

- producer/consumer problem

  producer

  shared memory buffer:
  - finite size
  - usual implementation: ring buffer

  consumer

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

- solve with semaphores
- is there a critical section?

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?
Producer/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
Semaphore mutex = 1;
Semaphore full = 0;
Semaphore empty = N;

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

consumer()
{
    int item;
    while(1)
    {
        wait(full);
        wait(mutex);
        remove_item(&item);
        signal(mutex);
        signal(empty);
        consume_item(&item);
    }
}
```
Ideal Solution?

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

Semaphore Solution to Producer Consumer

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

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

c consumer()
{
    int item;
    while(1) {
        wait(full);
        wait(mutex);
        remove_item(&item);
        signal(mutex);
        signal(empty);
        consume_item(&item);
    }
}
```
Ideal Solution?

- careless use of semaphores can lead to
  - deadlock
  - starvation if dequeueing 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 ⇒ 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)
Monitors

Monitors are used for synchronization in processes. They provide a way to ensure that multiple processes can access shared resources without interfering with each other.

In the diagram:
- **$P_A$**, **$P_B$**, **$P_C$**, and **$P_D$** represent processes.
- **init** is the initial state.
- The queue of entering processes is shown as a list of processes waiting to enter the monitor.
- The local data includes variables such as $c_0$, $c_1$, $c_2$, $c_{m-1}$, and $urg$.
- The executable procedures include the `append` and `take` functions.

Using monitors

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

    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);
    }
}
/* end monitor */

nextin = nextout = count = 0;

/* cont'd next page... */
```
Using monitors

```c
monitor

void producer()
{
    char x;
    while(1) {
        produce(&x);
        append(x);
    }
}

void consumer()
{
    char y;
    while(1) {
        take(y);
        consume(&y);
    }
}

void main()
{
    parbegin(producer,consumer);
}

/* from example in Stallings, © Prentice Hall 2001 */
```

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

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

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

- with cnotify():
  ```c
  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

- thread may stop running in method because
  - it finished
  - it can’t go forward, and must \texttt{wait}(): is placed in wait 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?

Thread Synchronization in Java

- 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 works 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 scenarios

1. blocking send and blocking receive
2. non-blocking send and blocking receive
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

what happens to blocked receive if:
(1) sender fails to send msg?
(2) msg is lost?
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:**
  - no direct communication between sending and receiving process
  - use common data structure as intermediary
  - these structures have queues for msgs and may be shared by many processes
    - a.k.a. mailboxes
  - sender sends to a mailbox, receiver picks up from a mailbox
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

---

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
### What messages look like

<table>
<thead>
<tr>
<th>vers</th>
<th>len</th>
<th>service type</th>
<th>total length</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ident</td>
<td></td>
<td>flags</td>
<td>fragment</td>
</tr>
<tr>
<td>time</td>
<td></td>
<td>proto</td>
<td></td>
</tr>
<tr>
<td>source address</td>
<td></td>
<td>destination address</td>
<td>padding</td>
</tr>
<tr>
<td>options</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>message data</td>
<td></td>
<td></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);
    }
}

int capacity=N; /* size of buffer */

void main() {
    create_mailbox(mayproduce);
    create_mailbox(mayconsume);
    for(i = 0; i < capacity; i++) send(mayproduce,null);
    parbegin(producer,consumer);
}
```

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