Concurrent Programming Problems

OS

Spring 2011

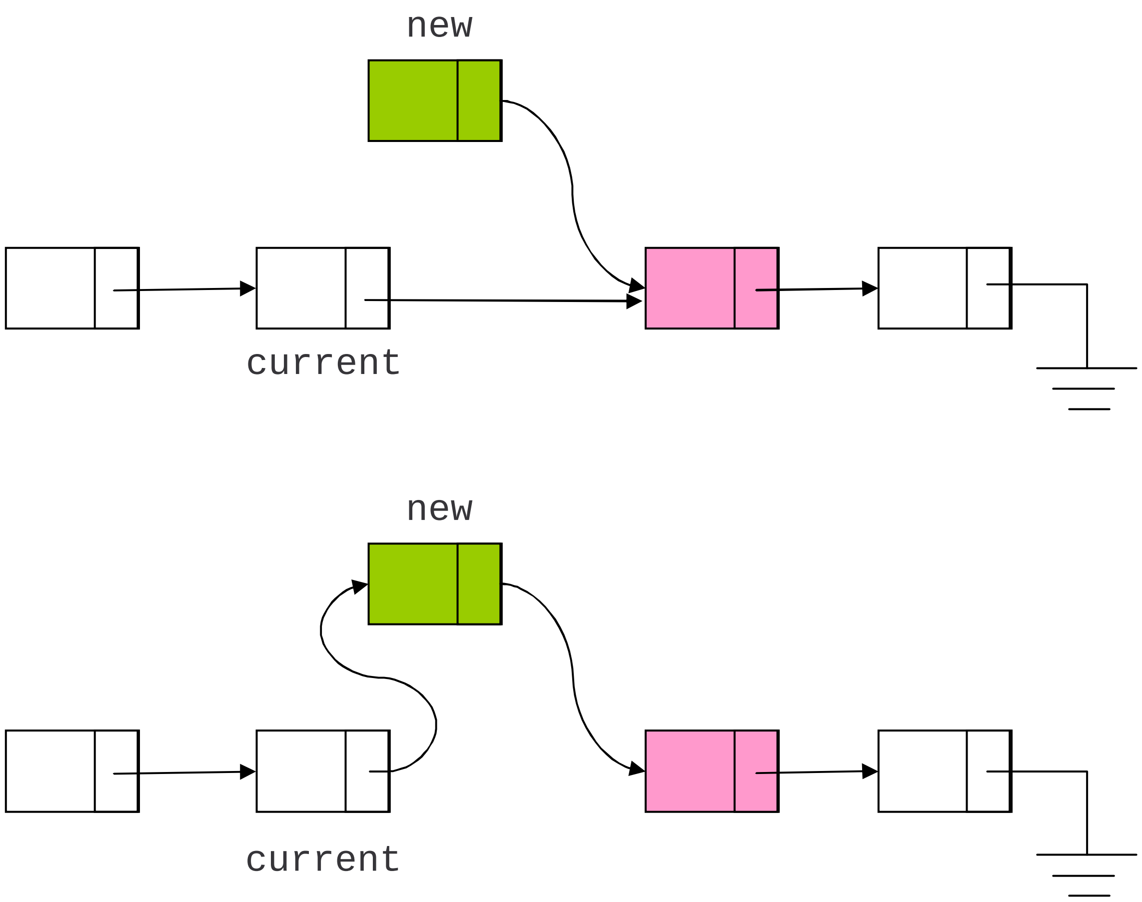
Concurrency pros and cons

- Concurrency is good for users
  - One of the reasons for multiprogramming
    - Working on the same problem, simultaneous execution of programs, background execution
- Concurrency is a “pain in the neck” for the system
  - Access to shared data structures
  - Danger of deadlock due to resource contention

Linked list example

insert_after(current,new): `new->next=current.next`

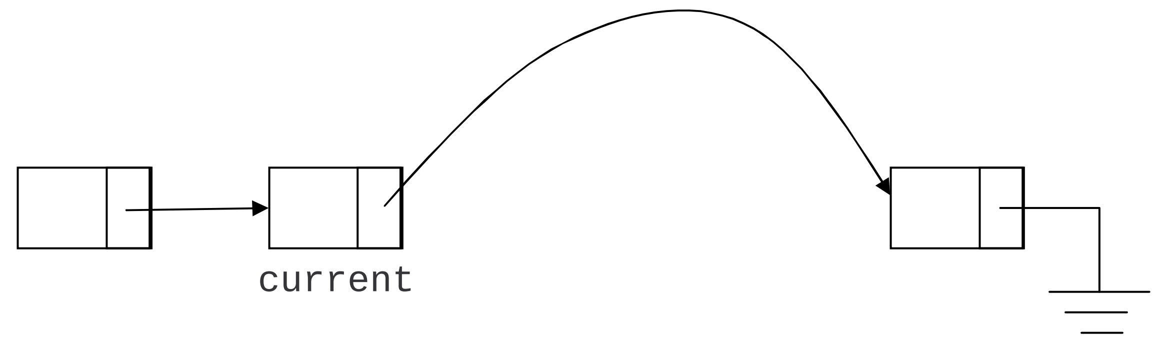
`current.next=new`



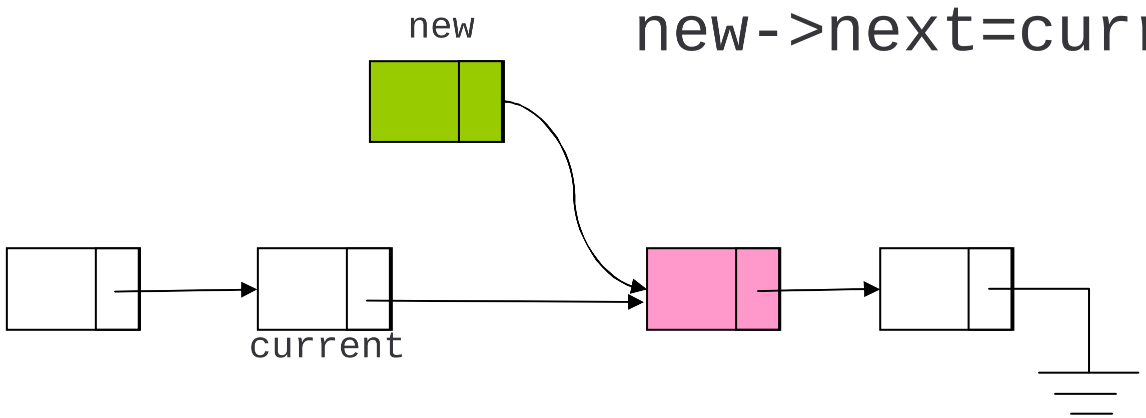
Linked list example

remove_next(current):

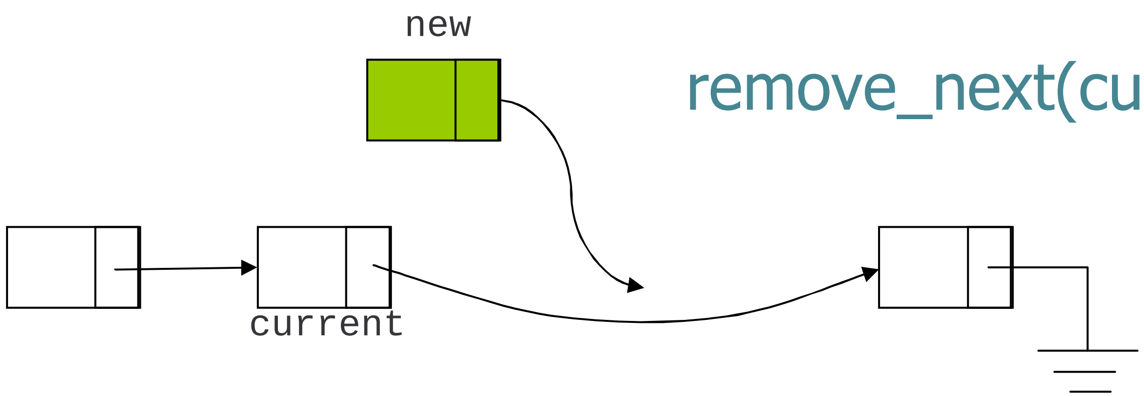
```
tmp=current.next;
current.next=current.next.next;
free(tmp);
```



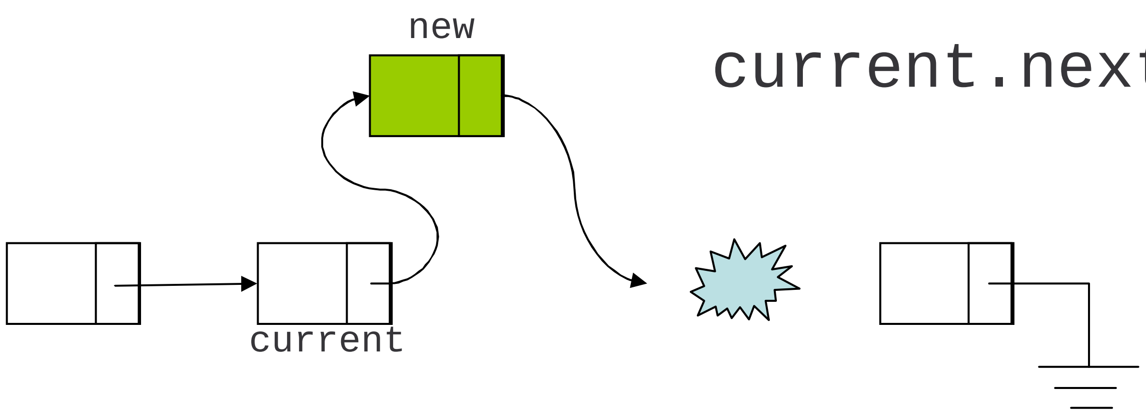
`new->next=current.next`



remove_next(current):



`current.next=new`



Mutual Exclusion

- OS is an instance of concurrent programming
  - Multiple activities may take place at the ‘same time’
- Concurrent execution of operations involving multiple steps is problematic
  - Example: updating linked list
- Concurrent access to a shared data structure must be *mutually exclusive*

Atomic Operations

- A generic solution is to ensure *atomic* execution of operations
  - All the steps are *perceived* as executed in a single point of time

insert_after(current,new) remove_next(current), or

remove_next(current) insert_after(current,new)

The Critical Section Problem

- n processes $P_0,\ldots,P_{n-1}$
- No assumptions on relative process speeds, no synchronized clocks, etc…
  – Models inherent non-determinism of process scheduling
- No assumptions on process activity when executing within critical section and remainder sections
- The problem: Implement a general mechanism for entering and leaving a critical section.

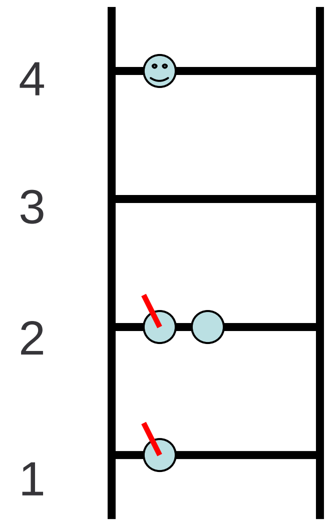
Success Criteria

1. *Mutual exclusion*: Only one process is in the critical section at a time.
2. *Progress*: There is no deadlock: some process will eventually get into the critical section.
3. *Fairness*: There is no starvation: no process will wait indefinitely while other processes continuously enter the critical section.
4. *Generality*: It works for N processes.

Peterson's Algorithm

- There are two shared arrays, initialized to 0.
  - q[n]: q[i] is the stage of process i in entering the CS. Stage n means the process is in the CS.
  - turn[n-1]: turn[j] says which process has to wait in stage j.
- The algorithm for process i:

```
for (j=1; j<n; j++) {
   q[i] = j;
   turn[j] = i;
   while ((∃k ≠i s.t. q[k]≥j)
           && (turn[j] == i)) {
        skip;
   }
}
critical section
q[i] = 0;
```



Proof of Peterson’s algorithm

```
for (j=1; j<n; j++) {
   q[i] = j;
   turn[j] = i;
   while((∃k ≠i st q[k]≥j)
        && (turn[j] == i)) {
     skip;
   }
}
critical section
q[i] = 0;
```

Definition: Process *a* is **ahead of** process *b* if
q[a] > q[b].

Lemma 1:

A process that is ahead of all others advances to the next stage (increments q[i]).

Proof: This process is not stuck in the while loop because the first condition does not hold.

Proof of Peterson’s algorithm

```
for (j=1; j<n; j++) {
   q[i] = j;
   turn[j] = i;
   while((∃k ≠i st q[k]≥j)
       && (turn[j] == i)) {
    skip;
   }
}
critical section
q[i] = 0;
```

Lemma 2:

When a process advances to stage j+1, if it is not ahead of all processes, then there is at least one other process at stage j.

Proof:

To exit the while loop another process had to take turn[j].

Proof of Peterson’s algorithm

```
for (j=1; j<n; j++) {
   q[i] = j;
   turn[j] = i;
   while((∃k ≠i st q[k]≥j)
        && (turn[j] == i)) {
     skip;
   }
}
critical section
q[i] = 0;
```

Lemma 3:

If there is more than one process at stage j, then there is at least one process at every stage below j.

Proof:

Use lemma 2, and prove by induction on the stages.

Proof of Peterson’s algorithm

```
for (j=1; j<n; j++) {
   q[i] = j;
   turn[j] = i;
   while((∃k ≠i st q[k]≥j)
        && (turn[j] == i)){
     skip;
   }
}
critical section
q[i] = 0;
```

Lemma 4:

The maximal number of processes at stage j is n-j+1

Proof:

By lemma 3, there are at least j-1 processes at lower stages.

Proof of Peterson’s algorithm

- Mutual Exclusion:
  - By Lemma 4, only one process can be in stage n
- Progress:
  - There is always at least one process that can advance:
    - If a process is ahead of all others it can advance
    - If no process is ahead of all others, then there is more than one process at the top stage, and one of them can advance.
- Fairness:
  - A process will be passed over no more than n times by each of the other processes (proof: in the notes).
- Generality:
  - The algorithm works for any n.

Peterson’s Algorithm

- Peterson’s algorithm creates a critical section mechanism without any help from the OS.
- All the success criteria hold for this algorithm.
- It does use busy wait (no other option).

Classical Problems of Synchronization

Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

Bounded-Buffer Problem

- One cyclic buffer that can hold up to N items
- **Producer** and **consumer** use the buffer
  - The buffer absorbs fluctuations in rates.
- The buffer is shared, so protection is required.
- We use **counting semaphores:**
  - the number in the semaphore represents the number of resources of some type

Bounded-Buffer Problem

- Semaphore mutex initialized to the value 1
  - Protects the index into the buffer
- Semaphore full initialized to the value 0
  - Indicates how many items in the buffer are full (can read them)
- Semaphore empty initialized to the value N
  - Indicates how many items in the buffer are empty (can write into them)

Bounded-Buffer Problem – Cont.

Producer:

```
while (true)  {
     produce an item
     P (empty);
     P (mutex);
          add the item to the  buffer
     V (mutex);
     V (full);
}
```

Consumer:

```
while (true) {
     P (full);
     P (mutex);
          remove an item from buffer
     V (mutex);
     V (empty);
     consume the item
}
```

Readers-Writers Problem

- A data structure is shared among a number of concurrent processes:
  - Readers – Only read the data; They do not perform updates.
  - Writers – Can both read and write.

- Problem
  - Allow multiple readers to read at the same time.
  - Only one writer can access the shared data at the same time.
  - If a writer is writing to the data structure, no reader may read it.

Readers-Writers Problem: First Solution

- Shared Data:
  - The data structure
  - Integer readcount initialized to 0.
  - Semaphore mutex initialized to 1.
    - Protects readcount
  - Semaphore write initialized to 1.
    - Makes sure the writer doesn’t use data at the same time as any readers

Readers-Writers Problem: First solution

- The structure of a **writer** process:

```
while (true) {
        P (write) ;

            writing is performed

        V (write) ;
}
```

Readers-Writers Problem: First solution

- The structure of a **reader** process:

```
while (true) {
        P (mutex) ;
              readcount ++ ;
              if (readcount == 1)
                 P (write) ;
        V (mutex)
              reading is performed
        P (mutex) ;
              readcount - - ;
              if (readcount  == 0)
                   V (write) ;
         V (mutex) ;
 }
```

Readers-Writers Problem: First solution

- The structure of a **reader** process:

```
while (true) {
        P (mutex) ;
              readcount ++ ;
              if (readcount == 1)
                 P (write) ;
        V (mutex)
             reading is performed
        P (mutex) ;
              readcount - - ;
              if (readcount  == 0)
                   V (write) ;
         V (mutex) ;
 }
```

This solution is not perfect:

What if a writer is waiting to write but there are readers that read all the time?

Writers are subject to starvation!

Second solution: Writer Priority

- Extra semaphores and variables:
  - Semaphore read initialized to 1 – inhibits readers when a writer wants to write.
  - Integer writecount initialized to 0 – counts waiting writers.
  - Semaphore write_mutex initialized to 1 – controls the updating of writecount.
  - Semaphore mutex now called read_mutex
  - Queue semaphore used only in the reader

Second solution:Writer Priority

The writer:

```
while (true) {
    P(write_mutex)
        writecount++;   //counts number of waiting writers
        if (write_count ==1)
                P(read)
    V(write_mutex)

    P (write) ;
        writing is performed
     V(write) ;

    P(write_mutex)
        writecount--;
        if (writecount ==0)
                V(read)
    V(write_mutex)
}
```

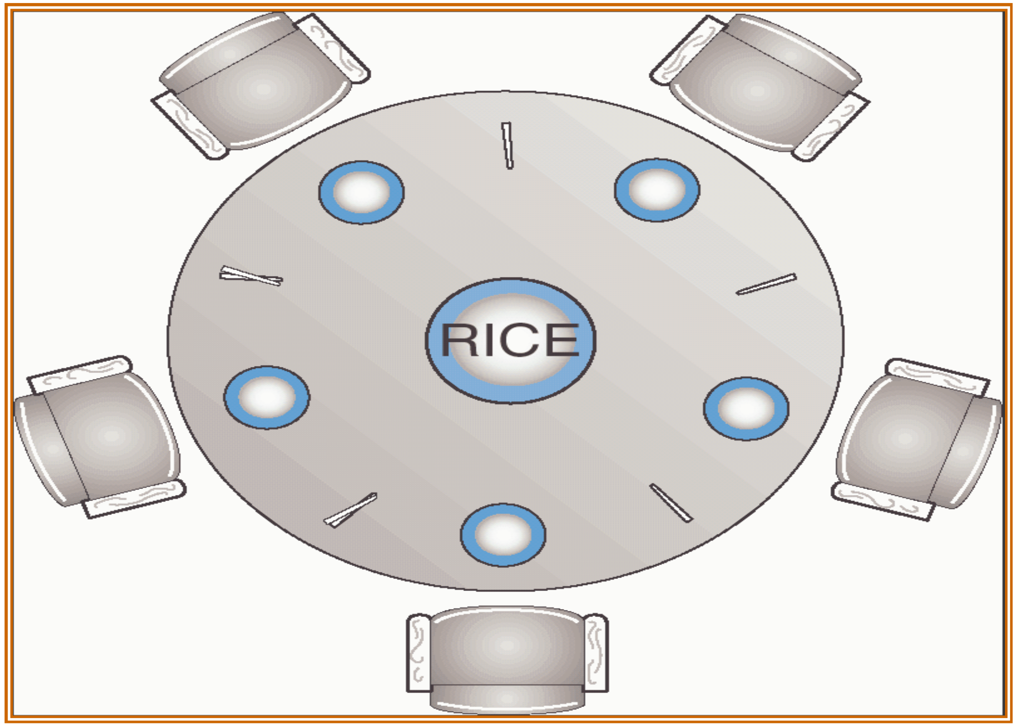
Second Solution: Writer Priority (cont.)

The reader:

```
while (true) {
      P(queue)
            P(read)
                P(read_mutex) ;
                      readcount ++ ;
                      if (readcount == 1)
                         P(write) ;
                V(read_mutex)
            V(read)
      V(queue)
      reading is performed
      P(read_mutex) ;
            readcount - - ;
            if (readcount  == 0)
               V(write) ;
      V(read_mutex) ;
}
```

Queue semaphore, initialized to 1: Since we don't want to allow more than one reader at a time in this section (otherwise the writer will be blocked by multiple readers when doing P(read).)

Dining-Philosophers Problem



Shared data

- Bowl of rice (data set)
- Semaphore chopstick [5] initialized to 1

Dining-Philosophers Problem – Cont.

- The structure of Philosopher i:

```
While (true)  {
   P ( chopstick[i] );
   P ( chopstick[ (i + 1) % 5] );

         eat

   V ( chopstick[i] );
   V (chopstick[ (i + 1) % 5] );

           think
}
```

- This can cause deadlocks ☹

Dining Philosophers Problem

- This abstract problem demonstrates some fundamental limitations of deadlock-free synchronization.
- There is no symmetric solution
  - Any symmetric algorithm leads to a symmetric output, that is everyone eats (which is impossible) or no-one does.

Possible Solutions

– Use a waiter
– Execute different code for odd/even
– Give them another chopstick
– Allow at most 4 philosophers at the table
– Randomized (Lehmann-Rabin)

Summary

- Concurrency can cause serious problems if not handled correctly.
  - Atomic operations
  - Critical sections
  - Semaphores and mutexes
  - Careful design to avoid deadlocks and livelocks.