Synchronization: Recap

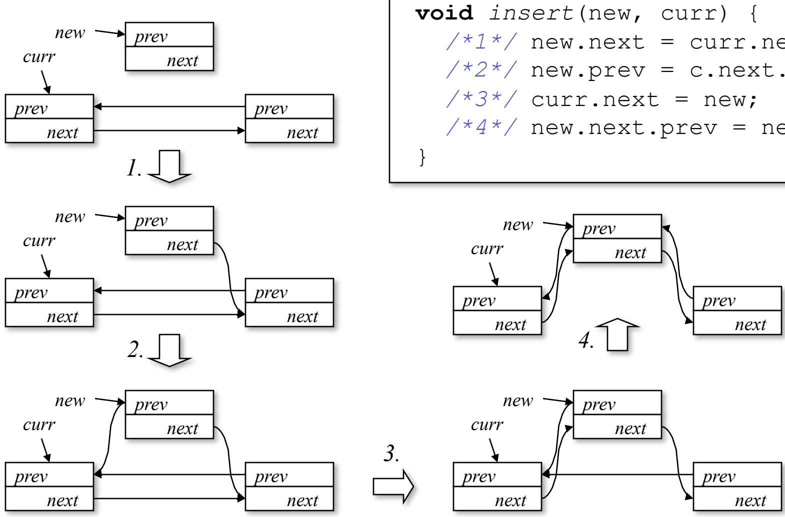
- Why?
  - Examples
- Semaphores
- Monitors

- Reading: Silberschatz, Ch. 6

Critical Section Problem: Example

Insertion of an element into a list.

```
void insert(new, curr) {
  /*1*/ new.next = curr.next;
  /*2*/ new.prev = c.next.prev;
  /*3*/ curr.next = new;
  /*4*/ new.next.prev = new;
}
```



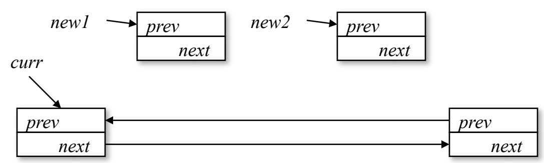
Interleaved Execution causes Errors!

Process 1

```
new1.next = curr.next;
new1.prev = c.next.prev;
…
…
…
…
curr.next = new1;
new.next.prev = new1;
```

Process 2

```
…
…
new2.next = curr.next;
new2.prev = c.next.prev;
curr.next = new2;
new.next.prev = new2;
…
…
```



- Must guarantee mutually exclusive access to list data structure!

Process Management: Synchronization

- **Why?** Examples
- **What?** The Critical Section Problem
- **How?** Software solutions
- Hardware-supported solutions
- The basic synchronization mechanism: Semaphores
- More sophisticated synchronization mechanisms: Monitors, Message Passing
- Classical synchronization problems

Hardware Support For Synchronization

- **Disallow interrupts**
  - simplicity
  - widely used
  - problem: interrupt service latency
  - problem: what about multiprocessors?
- **Atomic** operations:
  - Operations that check and modify memory areas in a single step (i.e. operation can not be interrupted)
  - **Test-And-Set**
  - **Exchange, Swap, Compare-And-Swap**

Test-And-Set

```
bool TestAndSet(bool & var) {
   bool temp;
   temp = var;
   var  = TRUE;
   return temp;
}
```

atomic!

Mutual Exclusion with *Test-And-Set*

```
bool lock; /* init to FALSE */

while (TRUE) {

  while (TestAndSet(lock)) no_op;

  critical section;

  lock = FALSE;

  remainder section;

}
```

Exchange (Swap)

```
void Exchange(bool & a, bool & b){
    bool temp;
    temp = a;
    a    = b;
    b    = temp;
}
```

atomic!

Mutual Exclusion with *Exchange*

```
bool lock; /*init to FALSE */

while (TRUE) {

  dummy = TRUE;
  do Exchange(lock, dummy);
  while(dummy);

  critical section;

  lock = FALSE;

  remainder section;
}
```

Compare-And-Swap

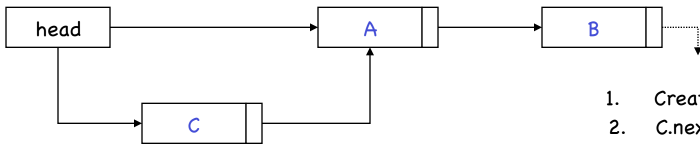
```
bool Compare&Swap (Type * x, Type old, Type new) {
      if *x == old {
            *x = new;
            return TRUE;
      } else  {
            return FALSE
      }
}
```

atomic!

Compare-and-Swap: Example
Lock-Free Concurrent Data Structures

Example: **Shared Stack**

PUSH element **C** onto stack:

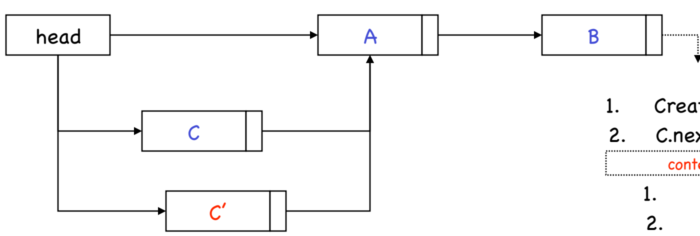


1. Create C
2. C.next = head
3. head = C

Compare-and-Swap: Example
Lock-Free Concurrent Data Structures

Example: **Shared Stack**

PUSH element **C** onto stack: What can go wrong?!



1. Create C
2. C.next = head

context switch!

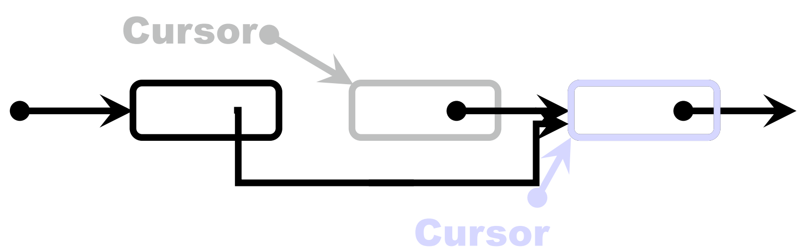
1. Create C'
2. C'.next = head
3. head = C'

context switch back!

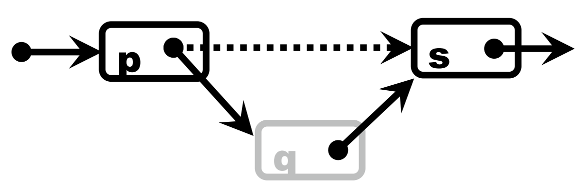
3. head = C

Solution: compare-and-swap(head, C.next, C),
i.e. compare and swap head, new value C, and expected value C.next.
If fails, go back to step 2.

Concurrent List Operations: Traversal



Concurrent List Operations: Insertion



Semaphores

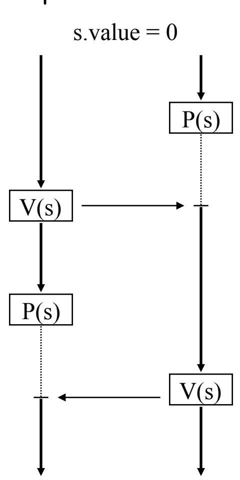
- Problems with solutions above:
  - Although requirements simple (mutual exclusion), addition to programs complex.
  - Based on busy waiting.
- A Semaphore variable has two operations:
  - **V(Semaphore * s);**

    /* Increment value of **s** by 1 in a single indivisible action. If value is not positive, then a process blocked by a **P** is unblocked*/
  - **P(Semaphore * s);**

    /* Decrement value of **s** by 1. If the value becomes negative, the process invoking the **P** operation is blocked. */
- **Binary semaphore**: The value of **s** can be either 1 or 0 (TRUE or FALSE).
- **General semaphore**: The value of **s** can be any integer.

Effect of Semaphores

- **General Synchronization** using semaphores:



- **Mutual exclusion** with semaphores:

```
BinSemaphore * s;
/* init to TRUE*/

while (TRUE) {

    P(s);

    critical section;

    V(s);

    remainder section;
}
```

Implementation (with busy waiting)

- Binary Semaphores:

```
P(BinSemaphore * s) {
  key = FALSE;
  do exchange(s.value, key);
  while (key == FALSE);
}

V(BinSemaphore * s) {
  s.value = TRUE;
}
```

- General Semaphores:

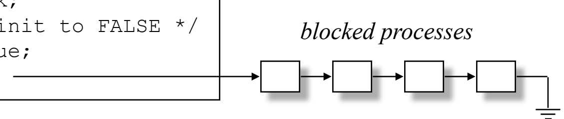
```
BinSemaphore * mutex /*TRUE*/
BinSemaphore * delay /*FALSE*/

P(Semaphore * s) {
 P(mutex);
 s.value = s.value - 1;
 if (s.value < 0)
   { V(mutex); P(delay); }
 else V(mutex);
}
V(Semaphore * s) {
 P(mutex);
 s.value = s.value + 1;
 if (s.value <= 0) V(delay);
 V(mutex);
}
```

Implementation (“without” busy waiting)

Semaphore

```
bool        lock;
            /* init to FALSE */
int         value;
PCBList * L;
```




```
P(Semaphore * s) {
 while (TestAndSet(lock))
   no_op;
 s.value = s.value - 1;
 if (s.value < 0) {
   append(this_process, s.L);
   lock = FALSE;
   sleep();
 }
 lock = FALSE;
}
```

```
V(Semaphore * s) {
 while (TestAndSet(lock))
   no_op;
 s.value = s.value + 1;
 if (s.value <= 0) {
   PCB * p = remove(s.L);
   wakeup(p);
 }
 lock = FALSE;
}
```

Classical Problems: **Producer-Consumer**

```
Semaphore    * n;     /* initialized to 0    */
BinSemaphore * mutex; /* initialized to TRUE */
```

```
Producer:

while (TRUE) {

  produce item;

  P(mutex);

  deposit item;

  V(mutex);
  V(n);

}
```

```
Consumer:

while (TRUE) {

  P(n);
  P(mutex);

  remove item;

  V(mutex);

  consume item;

}
```

Classical Problems: **Producer-Consumer with Bounded Buffer**

```
Semaphore    * full;  /* initialized to 0    */
Semaphore    * empty; /* initialized to n    */
BinSemaphore * mutex; /* initialized to TRUE */
```

```
Producer:

while (TRUE) {

  produce item;

  P(empty);
  P(mutex);

  deposit item;

  V(mutex);
  V(full);

}
```

```
Consumer:

while (TRUE) {

  P(full);
  P(mutex);

  remove item;

  V(mutex);
  V(empty);

  consume item;

}
```

Classical Problems: Readers/Writers

- Multiple readers can access data element concurrently.
- Writers access data element exclusively.

```
Semaphore * mutex, * wrt;  /* initialized to 1 */
int         nreaders;      /* initialized to 0 */
```

Reader:

```
P(mutex);
  nreaders = nreaders + 1;
  if (nreaders == 1) P(wrt);
V(mutex);

do the reading ....

P(mutex);
  nreaders = nreaders - 1;
  if (nreaders = 0) V(wrt);
V(mutex);
```

Writer:

```
P(wrt);

do the writing ...

V(wrt);
```

Monitors (Hoare / Brinch Hansen, 1973)

- Safe and effective sharing of abstract data types among several processes.
- Monitors can be modules, or objects.
  - local variable accessible only through monitor's procedures
  - process can entrer monitor only by invoking monitor procedure
- Only one process can be active in monitor.
- Additional synchronization through **conditions** (similar to semaphores)

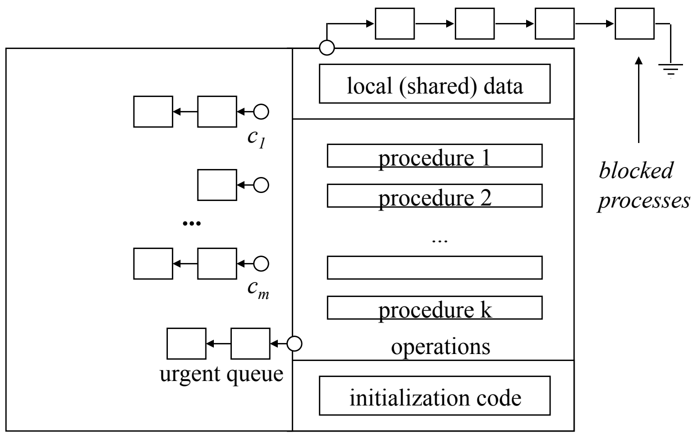
  `Condition c;`

  `c.cwait()` : suspend execution of calling process and enqueue it on condition `c`. The monitor now is available for other processes.

  `c.csignal()` : resume a process enqueued on `c`. If none is enqueued, do nothing.

  - `cwait`/`csignal` different from `P`/`V`: `cwait` always waits, `csignal` does nothing if nobody waits.

Structure of Monitor



Example: Binary Semaphore

```
monitor BinSemaphore {

  bool        locked; /* Initialize to FALSE */
  condition   idle;

  entry  void P() {
       if (locked) idle.cwait();
       locked = TRUE;
  }

  entry void V() {
       locked = FALSE;
       idle.csignal();
  }
}
```

Example: Bounded Buffer Producer/Consumer

```
monitor boundedbuffer {
  Item      buffer[N];     /* buffer has N items   */
  int       nextin;        /* init to 0            */
  int       nextout;       /* init to 0            */
  int       count;         /* init to 0            */
  condition notfull;       /* for synchronization */
  condition notempty;
```

```
void deposit(Item x) {
  if (count == N)
    notfull.cwait();
  buffer[nextin] = x;
  nextin = nextin + 1 mod
  N;
  count  = count + 1;
  notempty.csignal();
}
```

```
void remove(Item & x) {
  if (count == 0)
    notempty.cwait();
  x = buffer[nextout];
  nextout = nextout + 1 mod N;
  count  = count - 1;
  notfull.csignal();
}
```

Incorrect Implementation of Readers/Writers

```
monitor ReaderWriter{
    int numberOfReaders = 0;
    int numberOfWriters = 0;
    boolean busy = FALSE;

    /* READERS */
    procedure startRead() {
      while (numberOfWriters != 0);
      numberOfReaders = numberOfReaders + 1;
    }
    procedure finishRead() {
      numberOfReaders = numberOfReaders - 1;
    }

    /* WRITERS */
    procedure startWrite() {
      numberOfWriters = numberOfWriters + 1;
      while (busy || (numberOfReaders > 0));
      busy = TRUE;
    };
    procedure finishWrite() {
      numberOfWriters = numberOfWriters - 1;
      busy = FALSE;
    };
  };
```

A Correct Implementation

```
monitor ReaderWriter{
    int numberOfReaders = 0;
    int numberOfWriters = 0;
    boolean busy = FALSE;
    condition okToRead, okToWrite;

    /* READERS */
    procedure startRead() {
      if (busy || (okToWrite.lqueue)) okToRead.wait;
      numberOfReaders = numberOfReaders + 1;
      okToRead.signal;
    }
    procedure finishRead() {
      numberOfReaders = numberOfReaders - 1;
      if (numberOfReaders = 0) okToWrite.signal;
    }

    /* WRITERS */
    procedure startWrite() {
      if (busy || (numberOfReaders > 0)) okToWrite.wait;
      busy = TRUE;
    };
    procedure finishWrite() {
      busy = FALSE;
      if (okToWrite.lqueue) okToWrite.signal;
      else                  okToRead.signal;
    };
  };
```

Synchronization in JAVA

- Critical sections:
  - `synchronized` statement
- Synchronized methods:
  - Only one thread can be in any synchronized method of an object at any given time.
  - Realized by having a single lock (also called monitor) per object.
- Synchronized static methods:
  - One lock per class.
- Synchronized blocks:
  - Finer granularity possible using synchronized blocks
  - Can use lock of any object to define critical section.
- Additional synchronization:
  - `wait()`, `notify()`, `notifyAll()`
  - Realized as methods for all objects

Java Synchronized Methods: vanilla Bounded Buffer Producer/Consumer

```
public class BoundedBuffer {
 Object[] buffer;
 int      nextin, nextout;
 Object   notfull, notempty;
 int      size;
 int      count;
```

```
synchronized public deposit(Object x){
  if (count == size) notfull.wait();
  buffer[nextin] = x;
  nextin = (nextin+1) mod size;
  count  = count + 1;
  notempty.notify();
}
```

```
public BoundedBuffer(int n) {
  size    = n;
  buffer  = new Object[size];
  nextin  = 0;
  nextout = 0;
  count   = 0;
}
```

```
synchronized public Object remove() {
  Object x;
  if (count == 0) notempty.wait();
  x = buffer[nextout];
  nextout = (nextout+1) mod size;
  count  = count - 1;
  notfull.notify();
  return x;
}
```

Example: Synchronized Block

(D. Flanagan, *JAVA in a Nutshell*)

```
public static void SortIntArray(int[] a) {
  // Sort array a.  This is synchronized so that
  // some other thread cannot change elements of
  // the array or traverse the array while we are
  // sorting it.
  // At least no other thread that protect their
  // accesses to the array with synchronized.

  // do some non-critical stuff here...

  synchronized (a) {
     // do the array sort here.
  }

  // do some other non-critical stuff here...

}
```

Message Passing

- The Primitives:

```
send(destination, message);
receive(source, message);
```

- Issues:
  - Synchronization (blocking vs non-blocking primitives)
  - Addressing (direct vs. indirect communication)
  - Reliability / Ordering (reliable vs. unreliable)