Synchronization

Agenda

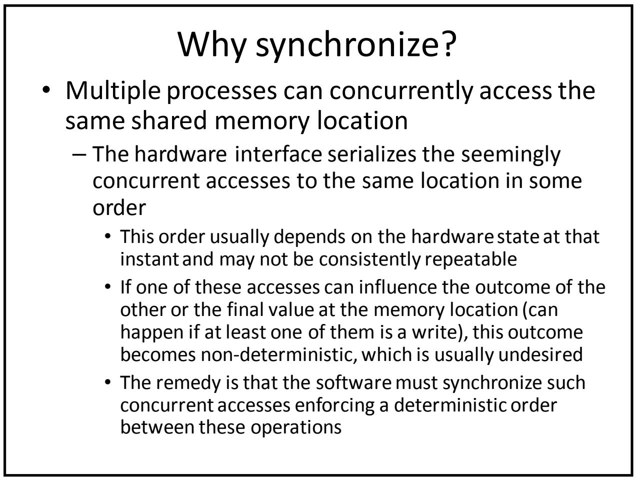
- Why synchronize?
- Critical sections and consensus
- Mutual exclusion algorithms
- Hardware support
- Semaphores
- Classical problems
- Monitors and condition variables
- POSIX synchronization interface

Why synchronize?

- Communicating processes or threads exchange data through messages and shared memory
  - In UNIX message passing, the data exchange essentially happens through a shared memory region in the kernel
    - The message queue is allocated in this region
    - Sender copies a message from its local address space to this kernel region
    - Receiver copies a message from this kernel region into its local address space
  - Shared memory regions attached to user address space avoid these costly copy operations

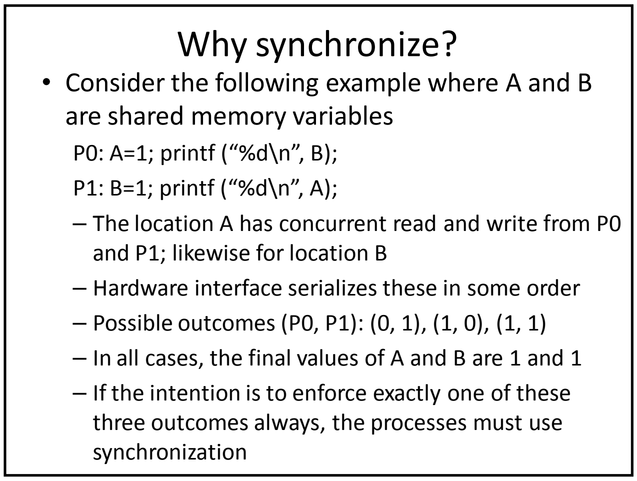
Why synchronize?

- Multiple processes share a region of memory by mapping the virtual addresses of this region to a common physical address
  - Processes A and B may have virtual addresses x and y corresponding to a shared variable, but this variable would have a unique physical address p
  - Virtual address x in A and virtual address y in B both map to the same physical address p
  - This mapping is established by the page table



Why synchronize?

- Multiple processes can concurrently access the same shared memory location
  - The hardware interface serializes the seemingly concurrent accesses to the same location in some order
    - This order usually depends on the hardware state at that instant and may not be consistently repeatable
    - If one of these accesses can influence the outcome of the other or the final value at the memory location (can happen if at least one of them is a write), this outcome becomes non-deterministic, which is usually undesired
    - The remedy is that the software must synchronize such concurrent accesses enforcing a deterministic order between these operations



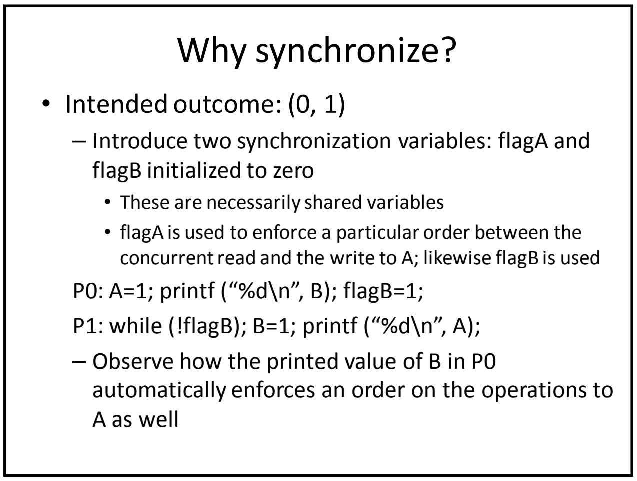
Why synchronize?

- Consider the following example where A and B are shared memory variables

  P0: A=1; printf (“%d\n”, B);

  P1: B=1; printf (“%d\n”, A);

  - The location A has concurrent read and write from P0 and P1; likewise for location B
  - Hardware interface serializes these in some order
  - Possible outcomes (P0, P1): (0, 1), (1, 0), (1, 1)
  - In all cases, the final values of A and B are 1 and 1
  - If the intention is to enforce exactly one of these three outcomes always, the processes must use synchronization



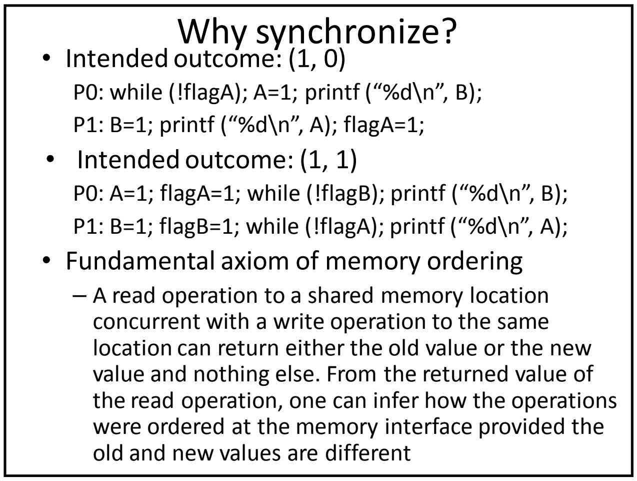
Why synchronize?

- Intended outcome: (0, 1)
  - Introduce two synchronization variables: flagA and flagB initialized to zero
    - These are necessarily shared variables
    - flagA is used to enforce a particular order between the concurrent read and the write to A; likewise flagB is used

  P0: A=1; printf (“%d\n”, B); flagB=1;

  P1: while (!flagB); B=1; printf (“%d\n”, A);

  - Observe how the printed value of B in P0 automatically enforces an order on the operations to A as well



Why synchronize?

- Intended outcome: (1, 0)

  P0: while (!flagA); A=1; printf (“%d\n”, B);

  P1: B=1; printf (“%d\n”, A); flagA=1;

- Intended outcome: (1, 1)

  P0: A=1; flagA=1; while (!flagB); printf (“%d\n”, B);

  P1: B=1; flagB=1; while (!flagA); printf (“%d\n”, A);

- Fundamental axiom of memory ordering
  - A read operation to a shared memory location concurrent with a write operation to the same location can return either the old value or the new value and nothing else. From the returned value of the read operation, one can infer how the operations were ordered at the memory interface provided the old and new values are different

Why synchronize?

- Vector sum example
  - Consider each location in the private_sum array
    - Each location is written to by a child and read from by the parent
    - We need to enforce the following ordering between the write and the read: the write must happen before the read
    - We could achieve this by attaching a flag with each array location; this is roughly what the wait() call achieves, since there is a one-to-one correspondence between the array location and the thread id
- This type of synchronization is called point-to-point synchronization and usually happens between a small group of threads (typically two)

Why synchronize?

- Consider the following example
  - Given a two-dimensional array A, we need to do the following computation repeatedly: A[i][j] = f(A[i][j])
    - The nature of f is such that |A[i][j] – f(A[i][j])| is monotonically decreasing with the number of iterations
    - The computation terminates when the total |A[i][j] – f(A[i][j])| accumulated over all elements of A falls below a threshold
    - One way to achieve this is to create a thread to handle the computation on A[i][j]
    - A thread can do its computation in iteration k provided all threads have completed the previous iteration and the total error is still greater than or equal to the threshold

Why synchronize?

- Need to order all operations in iteration k before all operations in iteration k+1
  - All threads must reach the end of an iteration before anybody is allowed to proceed further
  - Known as barrier synchronization, used to synchronize all threads globally
    - Intuitively, can be implemented by attaching a flag with each thread; each thread sets its own flag and waits on everybody else’s flag; must reset all flags after the barrier so that the next barrier instance can be executed
    - In reality, more thinking needed for a correct and efficient solution

Why synchronize?

- Consider the following example where x is a shared variable initialized to zero

  P0: x++;

  P1: x++;
  - Each process does a read, increment, write to x
  - Correct outcome requires that the read of P1 must be ordered after the write of P0 or the read of P0 must be ordered after the write of P1
  - Extending this to more than two processes, we require that the read-increment-write sequence from a process appears to be “atomic” i.e., there cannot be any operation to location x from any other process during this time

Why synchronize?

- Consider the following example where x is a shared variable initialized to zero

  P0: x++;

  P1: x++;

  – The exact order in which the sequences from the processes execute is irrelevant as long as each sequence is atomic
  – Such an atomic sequence is said to form a critical section
    - x++ is a critical section in this example
  – Different instances of the same critical section executing on different processes must be mutually exclusive in time

Critical sections and consensus

- Critical section is a code sequence that must be executed in a mutually exclusive manner by multiple processes/threads
  – Popularly known as an atomic region
- Common features of a critical section
  – Multiple threads may execute it concurrently
  – There is at least one write operation and at least one read to a shared memory location in the code sequence
  – Lot of examples in multithreaded user programs
  – Tricky to identify critical sections in OS code
    - Need to consider when kernel threads can be preempted
    - Inode list example

Critical sections and consensus

- Execution of critical section requires electing a leader or winner
  – A form of consensus or reaching agreement
  – Every critical section has three parts: entry to critical section, body of critical section, and exit from critical section
  – Entry section runs a consensus protocol
    - Will be referred to as a mutual exclusion protocol
  – Exit section prepares the states for the next round of consensus
    - Intuitively, resets the states to reflect that nobody is currently in the critical section

Critical sections and consensus

- Possible mutual exclusion protocols
  – Disable context switch on entry to critical section and re-enable on exit
    - Works for uniprocessor systems; fails on multiprocessors
    - May deliver poor performance
  – Make the OS kernel non-preemptive
    - Windows XP, Windows 2000, UNIX, Linux kernel before 2.6
    - A kernel mode thread/process cannot be switched out until it finishes the service at hand
    - All kernel data structures should be updated to a consistent state before beginning the physical I/O operation (not easy)
    - Pro: No need to figure out the critical sections in OS
    - Con: Not acceptable in real-time kernels
    - Con: In general, delivers poor performance

Critical sections and consensus

- Possible mutual exclusion protocols
  - What about user thread's critical sections?
  - Making the scheduler non-preemptive does not work
    - There could be an I/O operation inside a critical section e.g., reading a file, in which case a preemption may be necessary for performance reasons
- Need a generic solution that works in all environments with any kind of scheduler

Critical sections and consensus

- A technical term related to critical sections is "data races"
  - A data race is said to exist between two operations if
    - Both access the same memory location
    - They are executed by multiple independent threads/processes
    - At least one of them modifies the memory location
    - There exists at least one execution where one instruction executed by one thread appears adjacent to the other instruction executed by another thread with no synchronization operation in between
  - Data races are not good
    - Produces non-deterministic output and makes program debugging difficult
    - Critical sections with incorrect mutual exclusion protocols may lead to data races

Mutual exclusion algorithms

- A mutual exclusion algorithm defines the entry and exit protocols of a critical section
  - Comes in two flavors: purely software solutions and solutions relying on special hardware instructions to guarantee atomicity
- Three criteria of every good mutual exclusion algorithm
  - Mutual exclusion: At most one process can be inside the critical section at any point in time
  - Progress: If no process is executing in the critical section, exactly one of the waiting processes should be selected to enter the critical section and this selection cannot be postponed indefinitely
  - Bounded wait: The number of times other processes are allowed to enter the critical section from the time a process has expressed interest to enter the critical section should be bounded

Mutual exclusion algorithms

- Does the following work for two processes?

```
Precondition: i in {0, 1} and j = 1-i
Entry (i): flag[i] = 1;                // Express intent
           while (flag[j]);            // Wait until safe

Exit (i): flag[i] = 0;
```

- What happens if the two statements in entry are switched?
- Observation: somehow we need to order the processes in the entry section
  - This order is determined only at run-time and non-deterministic across different runs

Mutual exclusion algorithms

- Does the following work for two processes?

```
Precondition: i in {0, 1}, j = 1-i, turn in {0, 1}
Entry (i): while (turn == j);
Exit (i): turn = j;
```

- Imposes a static order (alternating) among the processes
- Fails when one of the processes disappears
  - The attempt in the last slide shows how to figure out if the other process wants to enter the critical section
  - Combine that with this to design a correct solution

Mutual exclusion algorithms

- A buggy attempt to combine the two

```
Precondition: i in {0, 1}, j = 1-i, turn in {0, 1}, flag[i] = 0
Entry (i): flag[i] = 1;
        // You’re there and your turn? I will wait then.
        while (flag[j] && (turn == j));

Exit(i): flag[i] = 0; turn = j;
```

–Problem case: this is the turn of process j, but j is not yet there. So process i enters (because flag[j] is false). Soon process j shows up and enters (because turn==j is false)

Dekker’s algorithm

- Does the following work for two processes?

```
Precondition: i in {0, 1}, j=1-i, turn in {0, 1}, flag[i]=0
Entry (i): flag[i] = 1;
        while (flag[j]) {
          if (turn == j) {
            flag[i] = 0;          // Be benevolent
            while (turn == j);
            flag[i] = 1;
          }
        }
Exit (i): turn = j; flag[i] = 0;
```

Dekker’s algorithm

- Observation: Cannot replace the outer while loop on flag[j] by an “if (flag[j])” statement
  - Consider a context switch between “while (turn==j);” and “flag[i]=1;” to generate a counterexample
- Observation: swapping the two statements in Exit(i) has no implication on correctness

Peterson’s algorithm

- How about the following?

```
Precondition: i in {0, 1}, j=1-i, and turn in {0, 1}
Entry (i): flag[i] = 1;
           turn = j;
           while (flag[j] && (turn == j));

Exit (i): flag[i] = 0;
```

- Observation: swapping “flag[i]=1;” and “turn=j;” leads to loss of mutual exclusion
  - Moving “turn=j;” to Exit(i) won’t work

Eisenberg and McGuire’s solution

- The first known n-process solution
  - Generalizes Dekker’s algorithm to n processes
  - The variable “turn” maintains who enters the critical section next
  - A process’s turn is decided by its id and the order in which it expresses intent relative to the other processes
  - The complexity of the protocol arises from the fact that there may be quiescent periods, at the end of which a group of processes may arrive together and turn may not point to any of these. One of these must be allowed to enter the critical section to satisfy progress

Lamport’s Bakery algorithm

- Bakery algorithm simplifies the Eisenberg-McGuire algorithm by making one observation
  - When a process expresses intent to enter a critical section, it takes a ticket (an integer) and waits for its turn to come
    - Ticket variable must be shared and a process may not get a unique ticket value
    - The algorithm must incorporate a tie-breaking rule if two tickets are identical: order such processes by their ids

Lamport’s Bakery algorithm

```
Precondition: ticket[i] = 0, choosing[i] = 0 for all i
Entry (i): choosing[i] = 1;
           ticket[i] = max (ticket[0], …, ticket[n-1])+1;
           choosing[i] = 0;
           for (j=0; j<n; j++) {
              while (choosing[j]);    // Why needed?
              while (ticket[j] && (ticket[j], j) < (ticket[i], i));
           }

Exit (i): ticket[i] = 0;
```

Hardware support

- Drawbacks of purely software locks
  - Doesn't offer a mechanical way for composing solutions
  - Solutions are usually complex and hard to verify
  - Complex solutions consume more CPU cycles if the critical section is frequently executed
    - For small critical sections, the critical section itself may consume less cycles than the mutual exclusion protocol
- Going back to the basics

```
Entry: while (lock); lock=1;    // want this read-write
                                // to be atomic
Exit: lock=0;               // This need not be atomic
```

Hardware support

- What if we had some instructions that could carry out a pair of load and store atomically
  - There are different classes of such memory operations, but each atomic instruction contains at least one load and one store to some memory location
  - TestAndSet is one such atomic instruction

    ts r, addr

    - This instruction brings the value at addr to register r and sets the value at addr to one; the entire operation looks like one atomic instruction i.e., no other instruction from any thread can access the location addr while the atomic instruction is in progress

Hardware support

- Mutual exclusion using TestAndSet

```
bool TAS (int *lockaddr) {
   asm ("ts x, lockaddr");  // Syntax not exact
   return x;
}

Entry: while (TAS (&lock));
Exit: lock=0;
```

Hardware support

- Another popular atomic instruction is exchange
  - It atomically exchanges the contents of two addresses

    xchg addr1, addr2

    - Essentially, involves two load and two store operations, all done atomically
    - Part of x86 ISA
- Another related atomic instruction is compare and exchange
  - Atomically compares the value at addr1 with an expected value V and if the comparison passes, it exchanges the contents of addr1 and addr2; otherwise it stores the contents of addr1 in addr2

Hardware support

- Mutual exclusion using xchg instruction

```
void XCHG (int *a, int *b) {
  asm (“xchg a, b”);
}

Entry: x=1; while (x) XCHG (&lock, &x);
Exit: lock=0;
```

Hardware support

- Mutual exclusion using cmpxchg instruction

```
void CAS (int Vexp, int *a, int *b) {
  asm (“cmpxchg Vexp, a, b”);
}

Entry: x=1; while (x) CAS (0, &lock, &x);
Exit: lock=0;
```

- While these algorithms are simple, they don’t guarantee bounded wait

Hardware support

- Mutual exclusion with hardware support and bounded wait

```
Precondition: key[i]=0 for all i
Entry (i): key[i] = 1;
            private flag=1;
            while (key[i] && flag) flag = TAS (&lock);
            key[i] = 0;

Exit (i): j = (i+1)%n;
         while ((j != i) && !key[j]) j = (j+1)%n;
         if (j == i) lock = 0;
         else key[j] = 0;
```

Semaphores

- Mutual exclusion or lock algorithms grant access to a number of processes into a critical section one at a time
- What if we want a bounded number of processes to access a resource simultaneously?
  - This is a form of synchronization
  - Example: consider a bounded buffer (a finite array); a number of producer processes can write new values into the array (provided the array is not full), which a number of consumer processes can read (provided the array is not empty)
  - Example: number of I/O buffers bounds the number of simultaneous I/O operations

Semaphores

- The bounded buffer problem
  - A naïve solution is to make the buffer access (read or write) a critical section
    - This is suboptimal because there is no reason to prevent multiple concurrent productions as long as there is room in the buffer; symmetrically, multiple concurrent consumptions should be allowed
  - N concurrent productions should be allowed if there are N empty slots in the buffer
  - N concurrent consumptions should be allowed if there are N new values in the buffer
  - Such resources are protected by counting semaphores (sometimes called mutexes)

Semaphores

- Counting semaphore
  - Has an integer value initialized to the maximum number of concurrent accesses to the resource it is protecting
    - The bounded buffer example would initialize the semaphore to k if the size of the buffer is k
- Special case where the integer can only take two values (usually zero and one) defines a binary semaphore
  - Binary semaphores can be used to implement locks

Semaphores

- In addition to the integer value, a semaphore offers two functions
  - wait and signal (or P and V from Dutch origin)

```
void wait (Semaphore S) {
  while (S.value <= 0);   // Busy-wait
   S.value--;
}

void signal (Semaphore S) {
  S.value++;
}
```

Semaphores

- Implementing a lock using binary semaphore

```
Initialization: Semaphore mutex=1;
Entry: wait (mutex);
Exit: signal (mutex);
```

  - Works provided wait and signal methods are atomic
    - Executing individual statements of wait atomically is not enough
    - Signal method needs to be atomic in non-binary semaphores
  - To make wait and signal atomic, we could resort to any mutual exclusion algorithm that we have discussed
    - Disabling context switch does not work with the current implementation of wait

Semaphores

- Current implementation of wait involves busy-waiting
  - Binary semaphores implemented in this way are called spinlocks
  - Wastes CPU cycles unnecessarily
    - Particularly important in uniprocessor systems
  - Better solution is to put the waiting processes in a queue and do a context switch
    - The signal call will walk this queue and wake up a waiting process (possibly the one at the head to ensure fairness)
    - Should the process be allowed to busy-wait for a while before doing the context switch? How long?
      - Makes sense if there are at least two processors so that there is a hope of breaking the busy-wait loop before doing a context-switch

Semaphores

- New implementation of wait

```
void wait (Semaphore S) {
  S.value--;
  if (S.value < 0) {
    EnqueueProcess (S.waitingQueue);
    context_switch();
  }
}
```

  - The magnitude of S.value (if it is negative) is the length of the waiting queue
    - One waiting queue per semaphore

Semaphores

- New implementation of signal

```
void signal (Semaphore S) {
  S.value++;
  if (S.value <= 0) {
     p = DequeueProcess (S.waitingQueue)
     EnqueueInReadyQueue (p);
  }
}
```

  - Which parts of wait and signal need to be atomic?

Semaphores

- More examples with semaphores for synchronizing two processes (these are not critical sections)
  - In all cases, s1 and s2 have initial values zero

```
P0: wait(s1); A=1; printf("%d\n", B);
P1: B=1; printf("%d\n", A); signal(s1);

P0: A=1; wait(s1); printf("%d\n", B);
P1: B=1; signal(s1); printf("%d\n", A);

P0: A=1; wait(s1); signal(s2); printf("%d\n", B);
P1: B=1; signal(s1); wait(s2); printf("%d\n", A);
```

Semaphores

- Consider the problem of dequeuing from one queue and enqueuing into another atomically
  - Must support concurrent execution by multiple processes
  - The queue object contains a linked list of items and a binary semaphore S initialized to one
  - The queue object supports two interface methods: Enqueue and Dequeue

```
void Enqueue (Item x) { Insert x in the list at the tail }
Item Dequeue (void) { Item x = head of list; move head;
return x; }
```

Semaphores

- Consider the following Move method to accomplish the task

```
void Move (Queue q1, Queue q2) {
   wait(q1.S); wait(q2.S);
   q2.Enqueue(q1.Dequeue());
   signal(q2.S); signal(q1.S);
}
```

  - Do you see any problem with this implementation?
    - Possible deadlock if process P0 does Move (Q1, Q2) and process P1 does Move (Q2, Q1) where Q1 and Q2 are queue objects

Synchronization problems

- Bounded buffer problem (buggy#1)
  - The producer-consumer problem on a buffer of size N
  - Two semaphores empty and full, initialized to N and 0
  - Variables nextp and nextc are initialized to 0

```
Producer:                        Consumer:
do { Generate a new item         do { wait(full);
       wait (empty);                    item = buffer[nextc];
       buffer [nextp] = item;           nextc = (nextc+1)%N;
       nextp = (nextp+1)%N              signal (empty);
       signal (full);                   Use item
} while (more to produce);       } while (more to consume);
```

Bounded buffer problem (Buggy#2)

```
Producer:                        Consumer:
Binary semaphore pro.v=1         Binary semaphore con.v=1
do { Generate a new item         do { wait (con);
       wait (pro);                      private index = nextc;
       private index = nextp;           nextc = (nextc+1)%N;
       nextp = (nextp+1)%N;             signal (con);
       signal (pro);                    wait (full);
       wait (empty);                    item = buffer[index];
       buffer[index] = item;            signal (empty);
       signal (full);                   Use item
} while (more to produce);       } while (more to consume);
```

Bounded buffer problem (Buggy#3)

```
Producer:
Binary semaphore pro.v=1
do { Generate a new item
      wait (empty);
      wait (pro);
      private index = nextp;
      nextp = (nextp+1)%N;
      signal (pro);
      buffer[index] = item;
      signal (full);
} while (more to produce);
```

```
Consumer:
Binary semaphore con.v=1
do { wait (con);
      private index = nextc;
      nextc = (nextc+1)%N;
      signal (con);
      wait (full);
      item = buffer[index];
      signal (empty);
      Use item
} while (more to consume);
```

Bounded buffer problem (Buggy#4)

```
Producer:
Binary semaphore pro.v=1
do { Generate a new item
      wait (empty);
      wait (pro);
      private index = nextp;
      nextp = (nextp+1)%N;
      signal (pro);
      buffer[index] = item;
      signal (full);
} while (more to produce);
```

Consumer:

- Initially buffer slots have a special value X

```
do { wait (full);
      private index = 0;
      while (buffer[index]==X)
            index++;
      item = buffer[index];
      buffer[index] = X;
      signal (empty);
      Use item
} while (more to consume);
```

Bounded buffer problem (Buggy#5)

- Initially buffer slots have a special value X

```
Producer:
do { Generate a new item
      wait (pro);
      …
      signal (pro);
      wait (empty);
      if (buffer[index]==X) {
        buffer[index] = item;
        signal (full);
      }
      else signal (empty);
} while (more to produce);
```

```
Consumer:
do { wait (full);
      private index = 0;
      while (buffer[index]==X)
            index++;
      item = buffer[index];
      buffer[index] = X;
      signal (empty);
      Use item
} while (more to consume);
```

Bounded buffer problem

- Fixing the last solution requires almost the entire producer and consumer to be atomic
  - Limits concurrency within the producers and within the consumers
  - We will take help of the atomic xchg and cmpxchg instructions to make only the minimally required portion atomic

Bounded buffer problem

```
Producer:
do { Generate a new item
     wait (pro);
     private index = nextp;
     nextp  = (nextp+1)%N;
     signal (pro);
     wait (empty);
     CAS (X, &buffer[index], &item);
     if (item == X) signal (full);
     else signal (empty);
} while (more to produce);
```

```
Consumer:
do {
  wait (full);
  private index = 0;
  private item = X;
  while (item == X) {
     XCHG (&buffer[index], &item);
     index++;
  }
  signal (empty);
  Use item
} while (more to consume);
```

Bounded buffer problem

- We can further simplify the producer

```
do {
   Generate a new item
   private index = 0;
   wait (empty);
   while (1) {
      original_item = item;
      CAS (X, &buffer[index], &item);
      if (item == X) break;
      index++;
      item = original_item;
   }
   signal (full);
} while (more to produce)
```

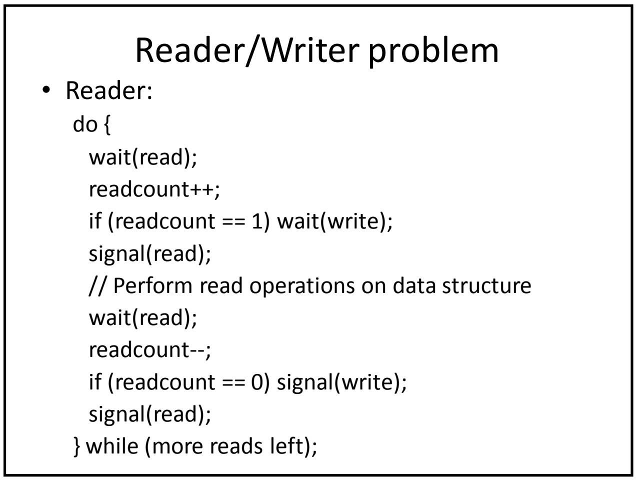
Reader/Writer problem

- Consider a data structure accessed by multiple reader and writer threads
  - Concurrent reads are allowed
  - We need to design a mutual exclusion protocol that allows
    - Multiple concurrent readers
    - No reader or writer if a writer is already accessing the data structure
    - No writer if at least one reader is already accessing the data structure

Reader/Writer problem

- Let write and read be two binary semaphores initialized to one
- Let readcount be initialized to zero

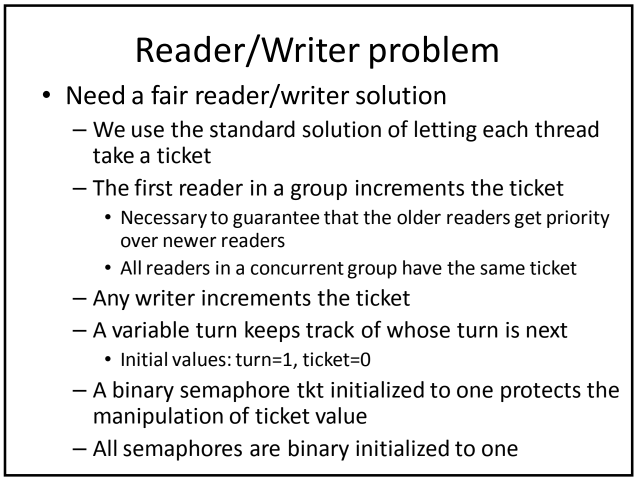
```
Writer:
do {
  wait(write);
  Perform write on data structure
  signal(write);
} while (more writes left);
```



Reader/Writer problem

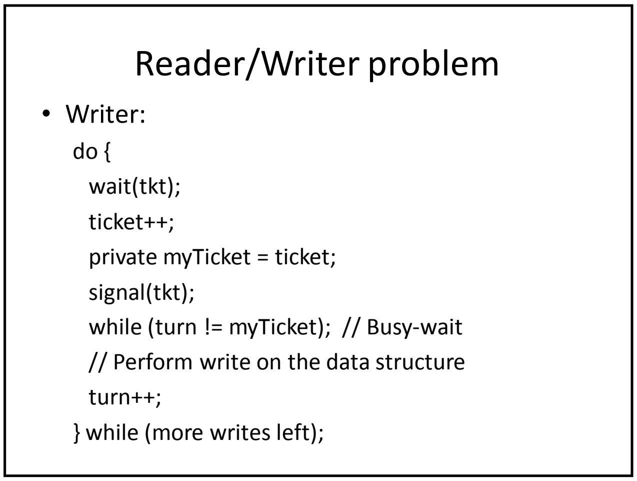
- Reader:

```
do {
  wait(read);
  readcount++;
  if (readcount == 1) wait(write);
  signal(read);
  // Perform read operations on data structure
  wait(read);
  readcount--;
  if (readcount == 0) signal(write);
  signal(read);
} while (more reads left);
```



Reader/Writer problem

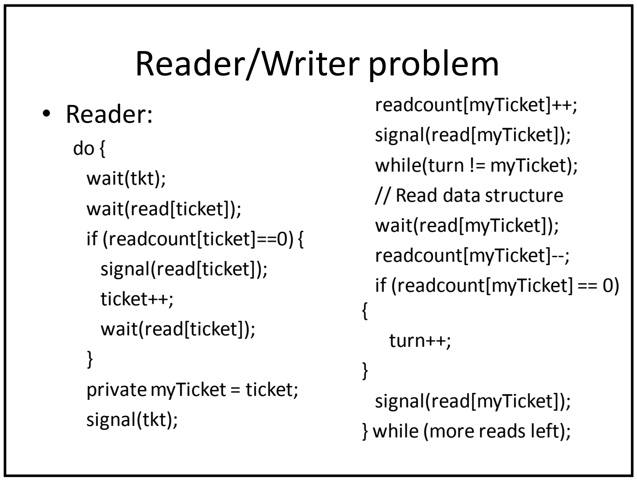
- Need a fair reader/writer solution
  - We use the standard solution of letting each thread take a ticket
  - The first reader in a group increments the ticket
    - Necessary to guarantee that the older readers get priority over newer readers
    - All readers in a concurrent group have the same ticket
  - Any writer increments the ticket
  - A variable turn keeps track of whose turn is next
    - Initial values: turn=1, ticket=0
  - A binary semaphore tkt initialized to one protects the manipulation of ticket value
  - All semaphores are binary initialized to one



Reader/Writer problem

- Writer:

```
do {
  wait(tkt);
  ticket++;
  private myTicket = ticket;
  signal(tkt);
  while (turn != myTicket);  // Busy-wait
  // Perform write on the data structure
  turn++;
} while (more writes left);
```



Reader/Writer problem

- Reader:

```
do {
  wait(tkt);
  wait(read[ticket]);
  if (readcount[ticket]==0) {
    signal(read[ticket]);
    ticket++;
    wait(read[ticket]);
  }
  private myTicket = ticket;
  signal(tkt);
  readcount[myTicket]++;
  signal(read[myTicket]);
  while(turn != myTicket);
  // Read data structure
  wait(read[myTicket]);
  readcount[myTicket]--;
  if (readcount[myTicket] == 0)
{
    turn++;
}
  signal(read[myTicket]);
} while (more reads left);
```

Dining philosopher problem

- Suppose n>1 philosophers are dining on a round table
  - To the left of every sitting position, there is a chopstick on the table
  - A philosopher can eat once she has both left and right chopsticks
  - How to synchronize the chopsticks?
  - Let chopstick[] be an array of binary semaphores, each initialized to one
  - How about the following solution?

```
wait(chopstick[i]); wait(chopstick[(i+1)%n]);
// Eat
signal(chopstick[(i+1)%n]); signal(chopstick[i]);
```

Dining philosopher problem

- Need a symmetry breaking rule
  - Central to deadlock avoidance
  - Let every even philosopher pick up her left chopstick first and every odd philosopher pick up her right chopstick first.

```
if ((i%2) == 0) { wait(chopstick[i]);
wait(chopstick[(i+1)%n]); }
else { wait(chopstick[(i+1)%n]); wait(chopstick[i]); }
// Eat
signal(chopstick[(i+1)%n]); signal(chopstick[i]);
```

- Any problem with this solution?

Sleeping barber problem

- Consider a barber shop with a waiting room containing n chairs
  - Barber usually sleeps until woken up by a customer
  - If a customer on arrival finds the barber to be busy with some other customer, he waits provided there is an empty chair in the waiting room; otherwise he leaves
  - How to synchronize the customers and the barber so that no customer waits unnecessarily?
  - Two binary semaphores chair and service, initialized to one and zero respectively
  - A counting semaphore barber, initialized to zero

Sleeping barber problem

- Customer

```
wait(chair);
myChair = chairCount;
if (chairCount < n) {
  chairCount++;
}
signal(chair);
if (myChair < n) {
  signal(barber);
  wait(service);
}
```

- Barber

```
do {
  wait(barber);
  signal(service);
  wait(chair);
  chairCount--;
  signal(chair);
} while (in business);
```

Monitors and condition variables

- Monitors are synchronization constructs that can encapsulate shared variables and methods
  - Similar to a class
  - The major difference from a class is that there can be at most one process at a time executing in any method inside a monitor
  - The monitor can include a special type of variable called a condition variable
    - A condition variable type supports three methods: wait, signal, and signalAll

Monitors and condition variables

- Condition variables
  - The wait method allows a process to wait on a condition variable
    - The implementation usually puts the caller to sleep
  - The signal method wakes up at most one process waiting on a condition variable
  - The signalAll method wakes up all the processes waiting on a condition variable
  - Does the caller of the signal or signalAll continue or undergo a context switch?
    - Signal and continue (lower overhead, but can be unfair)
    - Signal and wait (gives immediate quantum to a waiter)

Monitors and condition variables

- Condition variables
  - POSIX thread library allows the use of condition variables without any association with a monitor
    - POSIX does not even implement a monitor
    - Since condition variables are shared, they have to be accessed inside a critical section; without a monitor, condition variables are always associated with mutexes or semaphores or locks

Building monitors with semaphores

- Need to ensure mutual exclusion of every method
  - Every method that is a member of the monitor calls wait on a binary semaphore (say, mutex) initialized to one and calls signal just before returning from the method
    - Two scheduling options for selecting the next process: allow any process to enter the monitor or allow only one of the processes that is already inside the monitor
  - In addition, we need to implement the wait and signal methods for condition variables
    - We will assume signal and wait protocol
    - Every monitor contains a semaphore named benevolent initialized to zero for this purpose (in addition to mutex)

Building monitors with semaphores

- Need to ensure mutual exclusion of each method
  - A process completing a method call can choose one of the processes waiting after signal according to the second scheduling option of the last slide
  - We will assume that the subclass Condition implements condition variable objects
    - The Condition class contains a counting semaphore cond, initialized to zero used for implementing wait
    - Also contains an integer count to keep track of the number of processes waiting on the condition variable
  - A process before calling wait on cond wakes up any process waiting inside the monitor or allows a new process to enter the monitor by signaling mutex

Building monitors with semaphores

```
void Condition::wait ()
{
  count++;
  if (benevolentCount > 0) {
    signal(benevolent);
  }
  else signal(mutex);
  wait(cond);
  wait(mutex);
  count--;
}
```

```
void Condition::signal ()
{
  if (count > 0) {
    benevolentCount++;
    signal(cond);
    signal(mutex);
    wait(benevolent);
    benevolentCount--;
  }
}
```

Building monitors with semaphores

- Each method m of monitor M needs to be modified as follows

```
Type M::m (…)
{
  wait(mutex);
  Body of m without the return statement
  if (benevolentCount > 0) signal(benevolent);
  else signal(mutex);
  return statement
}
```

- Calling another method of this monitor leads to a deadlock
  - Need re-entrant locks (coming soon)

FIFO queue with CVs

```
void Enqueue (T x)
{
  wait(mutex);  // Initially 1
  while (count == size) {
     notFull.wait(mutex);
  }
  queue[tail] = x;
  tail = (tail+1)%size;
  count++;
  notEmpty.signal();
  signal(mutex);
}
```

```
T Dequeue (void)
{
  wait(mutex);
  while (count == 0) {
    notEmpty.wait(mutex);
  }
  T x = queue[head];
  head = (head+1)%size;
  count--;
  notFull.signal();
  signal(mutex);
}
```

FIFO queue with CVs

- The wait implementation of a condition variable signals the mutex guarding the condition variable before putting the caller to sleep
  - Otherwise nobody can make progress
- Before waking up such a process, a wait is executed on this mutex internally
- Pseudocode for wait:

```
Enqueue process in condition variable’s wait queue
Disable context switch
signal(mutex)
Change state of process to BLOCKED
Enable context switch
wait (mutex)
```

Counting semaphore from CVs

```
void CountSem::wait ()
{
  wait(mutex);   // Binary
  while (population == N) {
    empty.wait(mutex);
  }
  population++;
  signal(mutex);
}
```

```
void CountSem::signal ()
{
  wait(mutex);
  population--;
  empty.signal();
  signal(mutex);
}
```

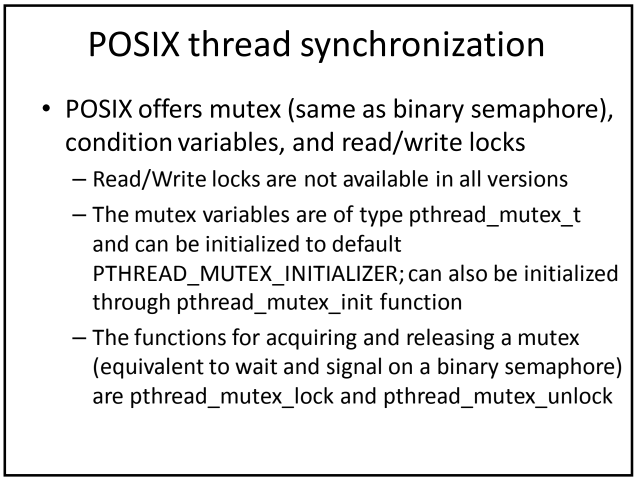
Reentrant locks

- The way we have designed locks or binary semaphores is not thread-safe
  - If a thread holding a lock attempts to acquire the same lock (i.e., calls wait before calling signal), it deadlocks
  - Often certain system functions hold a lock within them and recursive calls to such functions would lead to deadlock
  - Ideally, a thread should be able to acquire a lock multiple times before calling any signal
  - A lock is reentrant if it can be acquired by the same thread multiple times
    - The number of wait and signal calls must be balanced

Reentrant locks

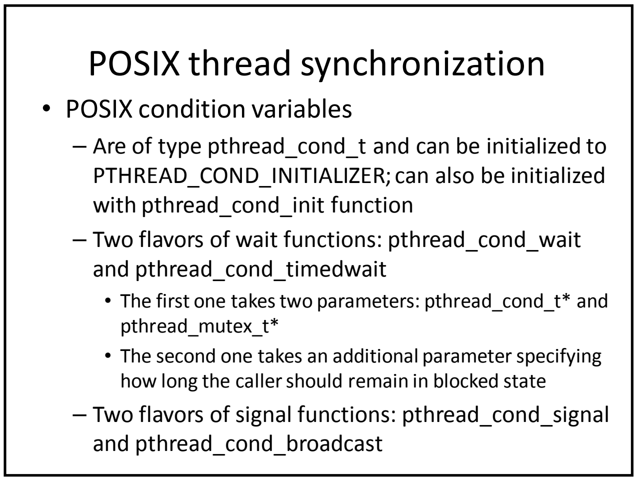
```
void ReentrantLock (void)
{
  wait(mutex); // Initially 1
  if (lockOwner == myID) {
    acquirer++;
  }
  else {
    while (acquirer != 0) {
      acquire.wait(mutex);
    }
    lockOwner = myID;
    acquirer = 1;
  }
  signal(mutex);
}
```

```
void ReentrantUnlock (void)
{
  wait(mutex);
  acquirer--;
  if (acquirer == 0) {
    acquire.signal();
  }
  signal(mutex);
}
```



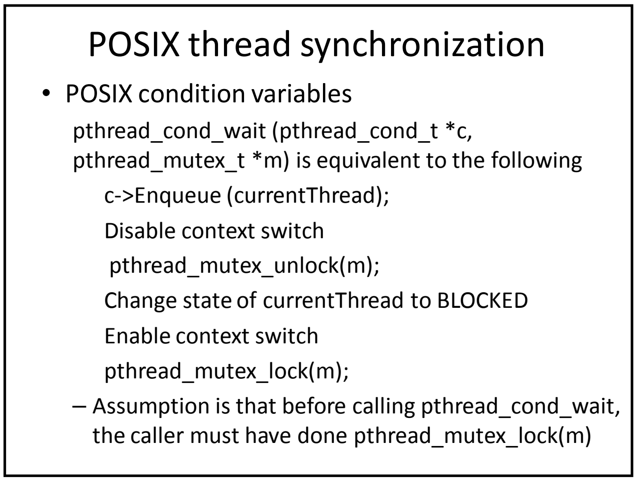
POSIX thread synchronization

- POSIX offers mutex (same as binary semaphore), condition variables, and read/write locks
  - Read/Write locks are not available in all versions
  - The mutex variables are of type pthread_mutex_t and can be initialized to default PTHREAD_MUTEX_INITIALIZER; can also be initialized through pthread_mutex_init function
  - The functions for acquiring and releasing a mutex (equivalent to wait and signal on a binary semaphore) are pthread_mutex_lock and pthread_mutex_unlock



POSIX thread synchronization

- POSIX condition variables
  - Are of type pthread_cond_t and can be initialized to PTHREAD_COND_INITIALIZER; can also be initialized with pthread_cond_init function
  - Two flavors of wait functions: pthread_cond_wait and pthread_cond_timedwait
    - The first one takes two parameters: pthread_cond_t* and pthread_mutex_t*
    - The second one takes an additional parameter specifying how long the caller should remain in blocked state
  - Two flavors of signal functions: pthread_cond_signal and pthread_cond_broadcast



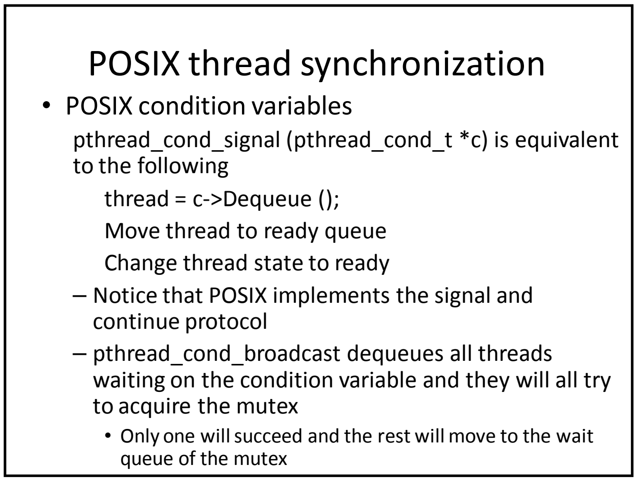
POSIX thread synchronization

- POSIX condition variables

pthread_cond_wait (pthread_cond_t *c, pthread_mutex_t *m) is equivalent to the following

```
c->Enqueue (currentThread);
Disable context switch
 pthread_mutex_unlock(m);
Change state of currentThread to BLOCKED
Enable context switch
pthread_mutex_lock(m);
```

  - Assumption is that before calling pthread_cond_wait, the caller must have done pthread_mutex_lock(m)



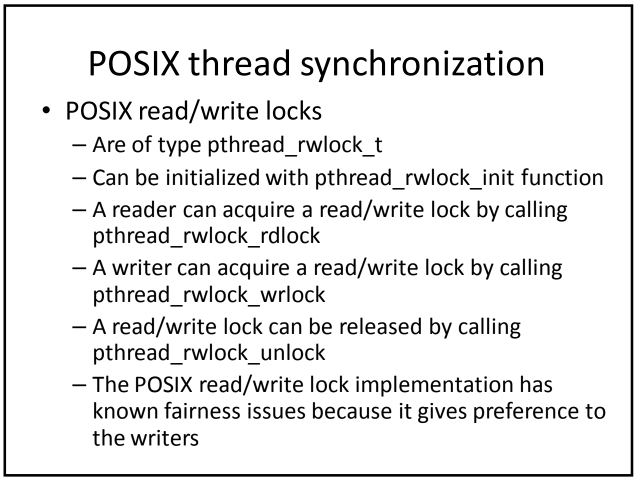
POSIX thread synchronization

- POSIX condition variables

pthread_cond_signal (pthread_cond_t *c) is equivalent to the following

```
thread = c->Dequeue ();
Move thread to ready queue
Change thread state to ready
```

  - Notice that POSIX implements the signal and continue protocol
  - pthread_cond_broadcast dequeues all threads waiting on the condition variable and they will all try to acquire the mutex
    - Only one will succeed and the rest will move to the wait queue of the mutex



POSIX thread synchronization

- POSIX read/write locks
  - Are of type pthread_rwlock_t
  - Can be initialized with pthread_rwlock_init function
  - A reader can acquire a read/write lock by calling pthread_rwlock_rdlock
  - A writer can acquire a read/write lock by calling pthread_rwlock_wrlock
  - A read/write lock can be released by calling pthread_rwlock_unlock
  - The POSIX read/write lock implementation has known fairness issues because it gives preference to the writers