Repetition

Eric Jul and Paulo Ferreira
Programming Technology Group
Programming Section
Department of Informatics
University of Oslo

1

Lectures – what you are supposed to know

2

Starting

- Intro
- There are multi-core machines that require concurrent programming
- There are some tasks that are inherently parallel and other that are not:
  - Plant 10000 trees
  - A woman produce a baby in 9 months
- Threads concept and how they work in Java

3

Efficiency

- Speedup - Efficiency
- Efficiency and correct
- It is really simple to write a program that is not correct

- Super linear speedup (is it possible to has 10 cores and have a speedup more than 10) – can you achieve that? there must be something wrong…
- Amdahl Law
- Cost of creating threads
- Oblig1 - Find the k largest numbers in a large array

4

Time Slicing

- Time slicing
- Concurrent reads are OK but concurrent writes are not
- Synchronization (semaphores, cyclic barrier, etc.):
  - Reads are OK
  - Concurrent writes are not !
- There will be nothing on quantum computing

5

Measuring

- Measuring programs (speed, wall clock time)
- This times are not the same (median will be used):
  - example of driving taking into account rush or not
- You can be hit by a garbage collection, Internet activity (big mail coming in), effects of caching, just in time compilation (JIT), pre-fetching – different from a few years ago!
- You should warm up !

6

Caching

- Why ?
  - Slow speed of light (299,792,458 m/s)
  - Trade-off speed vs size
- The meter is defined by the speed of light - meter was defined as the fraction of distance from the equator to the north pole
- How far does electricity propagate in copper in 1 ns ?
  - Light travels 6 cm (2 clock cycles)
- Multi–level caching
- Memory takes 200 cycles and L0 caching take 4 cycles (a factor of 50)
- A mechanism that is called prefetching
- Oblig2 matrix

7

Moore´s

- Moore´s law
  - It was valid until recently
  - Switch transistors is faster
- Performance leads to caching
- There are multi-cores CPU
- How to parallelize:
  - Split (different data for several threads)
  - Program pipeline
- Divide and Conquer: Convex Hull
- Economize the creation of threads
- Pool of threads (Java has this)

8

Sieve and Factorization

- Prime number oblig3
- You parallelize (an iterative) sieve and the factorization

9

Synchronization

- Sketch
- Andes trains with blind and deaf drivers
- Chefs and waiters:
  - Busy waiting
  - Action initiated
  - wait and signal
- Hoare (1973) and Mesa monitors
- Oblig4 convex

10

More on Synchronization

- Paralellization of Convex Hull
- Oblig5 synchronization
- Synchronization
- Race conditions (two threads want to grab something)-varispeed
- Travelings salesman method - Monte Carlo
  - The optimal solution takes a long time
  - How much improvement did I get the last time
- Multiple semaphores:
  - Problem: signal a semaphore (before waiting you have to release the lock)

11

Yet More on Synchronization

- Competition
- Typical problems of synchronization (producer consumer, philosophers ´ dinner, and barber shop)
- Condition variables

12

Competition

- A proc/thread demands a resource:
  - runs acquire (SemResource)
- A proc/thread frees a resource:
  - runs release (SemResource)
- The semaphore that controls the algorithm is initialized with the number of availabe resources:
  - SemResource = new Semaphore(NUM_RESOURCES)

16

Semaphores (example)

```
#define MAX_STACK 100
char*    stack[MAX_STACK];
int      top = MAX_STACK-1;
Semaphore SemMutEx;
Semaphore SemMem;

char *ask_for_mem() {
   acquire (SemMem);
   acquire (SemMutEx);
   ptr = stack[top];
   top--;
   release (SemMutEx);
   return ptr;
}

void get_back_mem() {
   acquire(SemMutEx);
   top++;
   stack[top] = ptr;
   release (SemMutEx);
   release (SemMem);
}
```

Memory Allocator with semaphore SemMem that controls the free memory !

```
main(){
/*...*/
   semExMut = new Semaphore(1);
   semMem = new Semaphore(MAX_STACK);
}
```

The semaphore is initialized with the number of available resources for the client

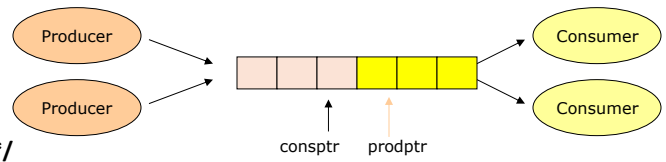
17

Typical Examples of Synchronization

- Producers/Consumers
  - Threads/procs that produce information to a buffer and threads/procs that read the information from that buffer (semaphores)
- Dining-Philosophers
  - Threads/procs that eat spaghetti and has less forks that eaters (semaphores)
- Barber
  - One or more threads/procs that serve the client thread (semaphores and conditions

18

Example: Producers - Consumers



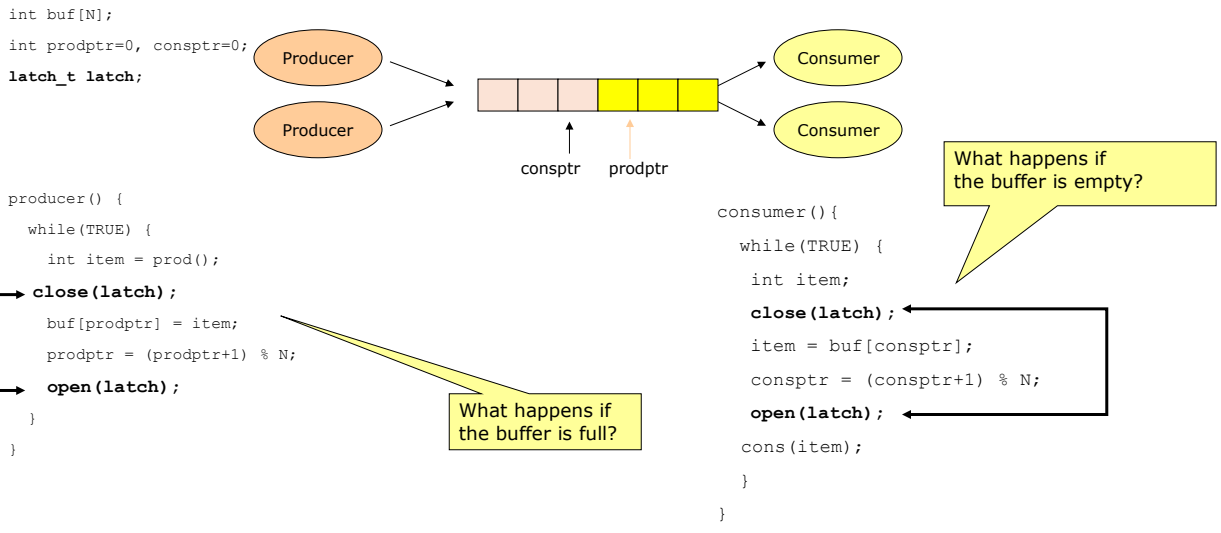

```
/* ProducerConsumer */
int buf[N];
int prodptr=0, consptr=0;

producer()
  while(TRUE) {
    int item = prod();
    buf[prodptr] = item;
    prodptr = (prodptr+1) % N;
  }
}
```

```
consumer(){
  while(TRUE) {
    int item;
    item = buf[consptr];
    consptr = (consptr+1) % N;
    cons(item);
  }
}
```

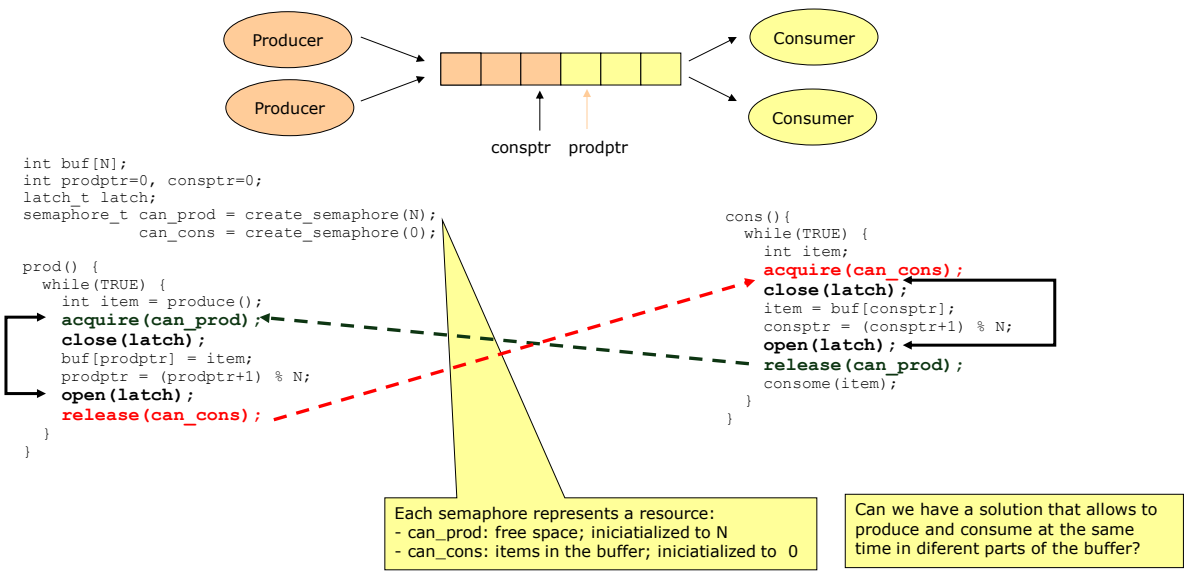
19

Example: Producers - Consumers



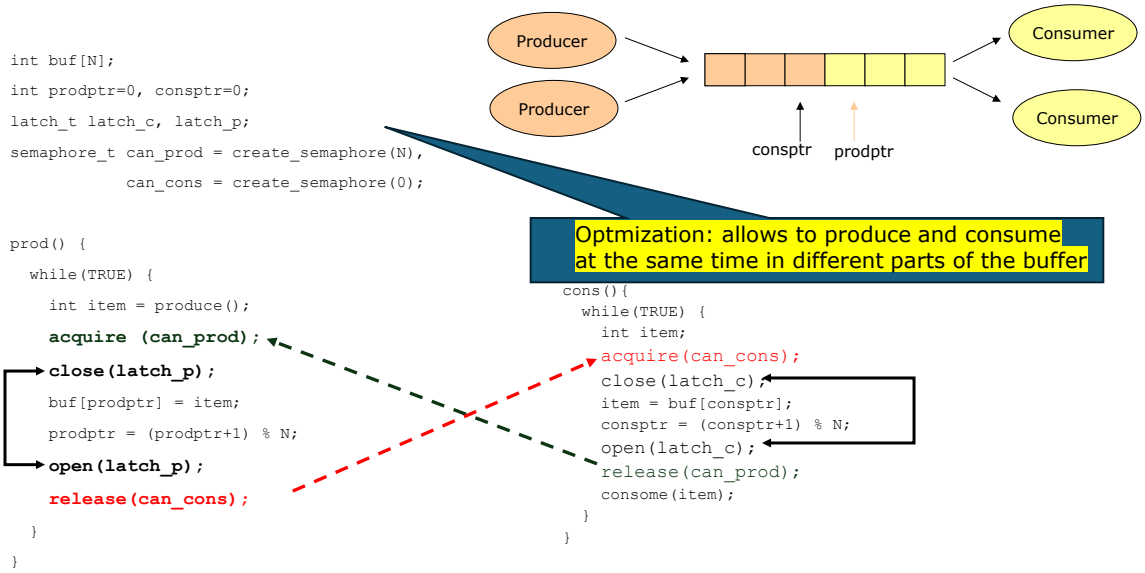
20

Example: Producers - Consumers



21

Example: Producers - Consumers




```
int buf[N];
int prodptr=0, consptr=0;
latch_t latch_c, latch_p;
semaphore_t can_prod = create_semaphore(N),
            can_cons = create_semaphore(0);

prod() {
  while(TRUE) {
    int item = produce();
    acquire (can_prod);
    close(latch_p);
    buf[prodptr] = item;
    prodptr = (prodptr+1) % N;
    open(latch_p);
    release(can_cons);
  }
}
```

```
cons(){
  while(TRUE) {
    int item;
    acquire(can_cons);
    close(latch_c);
    item = buf[consptr];
    consptr = (consptr+1) % N;
    open(latch_c);
    release(can_prod);
    consome(item);
  }
}
```

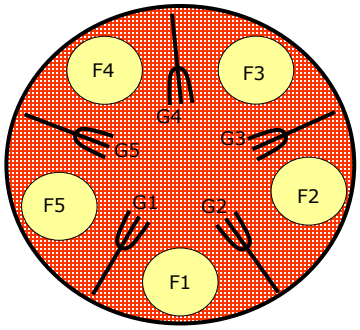
22

Philosophers Dinner

- Five philosophers are re-united to philosophate and to have dinner eating spaghetti:
  - To eat spaghetti each one needs two forks, but the table provides one fork per each
- Conditions:
  - The philosophers can be in one of the following three states: Think, Hungry, Eat.
  - Each place that a pilosopher occupies is fixed
  - Each pilosopher can only use the forks immediately on his left and right

23

Philosophers Dinner

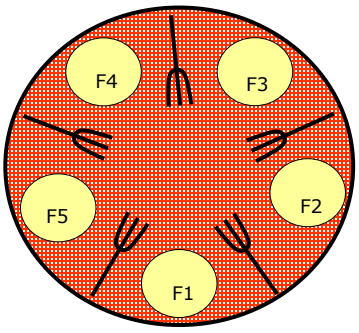


24

Philosophers Dinner

```
semaphore_t new Semaphore fork[5] = {1, 1, 1, 1, 1};

philosopher(int id)
{
    while (TRUE) {
       think();
       acquire(fork[id]);
       acquire(fork[(id+1)%5]);
       eat();
       release(fork[id]);
       release(fork[(id+1)%5]);
    }
}
```



25

Interblocking

- A situation of interblocking can occur if the four following conditions are simultaneously true:
  - At least one resource is used in a non-sharable way
  - There is at least one process that holds a resource and that is waiting to acquire mode resources
  - The resources are not “preemptible”, i.e., the resources are not voluntarily released by the process that hold them
  - There is a pattern of synchronization in which proc/thread T1 waits for a resource from T2 and circularly Tn+1 waits for a resource from T1

26

Interblocking– preventive solutions

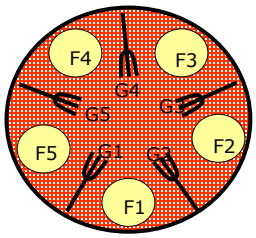
- Making sure that all the resources are acquired in the same order
- Acquiring all the resources when the thread starts
- When the acquisition of resource is not possible for the progression of the algorithm, releasing all the resources a thread holds

27

Philosophers Dinner with Semaphores, version #2

```
semaphore_t new Semaphore fork[5] =
   {1, 1, 1, 1, 1};

filosopher(int id)
{
    while (TRUE) {
        think();
        if (id%2 == 1)
          acquire(fork[(id+1)%5]);
          acquire(fork[id]);
        } else {
          acquire( fork[id]);
          acquire(fork[(id+1)%5]);
        }
        eat();
        release(fork[id]);
        release(fork[(id+1)%5]);
    }
}
```



- Always acquire the semaphores in the same order - *odd philosophers start left-hand first, and even philosophers start right-hand first*
- Other solutions:
  - Acquire resources at start of the execution with non-blocking call "trylock", release if it fails
  - Detection of interblocking: forced release by elimination of the proc/thread

28

Philosophers Dinner with Semaphores, version #3



29

Philosophers Dinner with Semaphores, version #3

```
#define THINK 0
#define HUNGRY 1
#define EAT 2
#define N 5
int state[N] = {0, 0, 0, 0, 0};
semaphore_t new Semaphore semphilo[N] =
      {0, 0, 0, 0, 0};
latch_t mutex;

Test(int k){
  if (state[k] == HUNGRY &&
      state[(k+1)%N] != EAT &&
      state[(k-1)%N] != EAT){
         state[k] = EAT;
         release(semphilo[K]);
  }
}
```

```
filosopher(int id)
{
    while (TRUE) {
        think();
        acquire(mutex);
        state[id] = HUNGRY;
        Test(id);
        release(mutex);
        acquire(semphilo[id]);
        eat();
        acquire(mutex);
        state[id] = THINK;
        Test((id-1+N)%N);
        Test((id+1)%N);
        release(mutex);
    }
}
```

- Use a semaphore to represent the conditions of blocking (and not the status of the forks/philosophers)
- Represent the status of the forks/philosophers with a set of variables that are accessed in a critical section

30

Philosophers Dinner with Semaphores, version #4

```
semaphore_t new Semaphore fork[5] =
    {1, 1, 1, 1, 1};
semaphore_t new Semaphore room = 4;

filosopher(int id)
{
    while (TRUE) {
        think();
        acquire(room);
        acquire(fork[id]);
        acquire(fork[(id+1)%5]);
        eat();
        release(fork[id]);
        release(fork[(id+1)%5]);
        release(room);
    }
}
```

- Limit the access to the room to N-1 filosophers (always be free at least a fork)

31

Barber

- In a barber shop there is a chair in which:
  - The barber cuts the hair
  - N (for clients) waiting chairs
- If there are no clients:
  - The barber sits on the chair and sleeps
- When a client arrives:
  - He must wake the sleeping barber for him to cut the client`s hair



- If, in the meantime, more client arrives, while the barber is cutting the hair to the first client:
  - They wait on a waiting chair, or
  - They leave, assuming there no more waiting chairs

32

Barber

```
void barber() {
  while(TRUE) {
    Acquire(clients);
    SeeClient();
    Acquire(mutex);
    clients_wating--;  /* remove a client */
    Release(barber); /* unblock a client */
    Release(mutex);
  }
}
```

```
#define MAX_CHAIRS N
latch_t mutex;
int clients_waiting= 0;
semaphore_t clients, barber;

main{ /*...*/
   mutex = new Semaphore(1);
   clients = new Semaphore(0);
   barber = new Semaphore(0);
/*...*/}
```

```
int customer() {
  Acquire(mutex);
  if (clients_waiting < MAX_CHAIRS) {
    client_waiting++;     /* add a client */
    Release(clients);     /* tell barber that there are clients */
    Release(mutex);
    Acquire(barber);      /* wait for the barber to be ready */
    HairCut();
  }
  else {
    Release(mutex);
    return 0;             /* give up if there are no chairs */
  }
```

33

Barber in Java

```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
import java.util.Random;

public class BarberShop {
    private final Lock lock = new ReentrantLock();
    private final Condition condition = lock.newCondition();
    private final int[] waiting = new int[3];
    private int waitingCount = 0;
    private static final int NUM_CHAIRS = 3;
    private Random random = new Random();

    public void barber() {
        while (true) {
            lock.lock();
            try {
                while (waitingCount == 0) {
                    try {
                        condition.await();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                int cust = waiting[0];
                for (int i = 0; i < waitingCount - 1; ++i) {
                    waiting[i] = waiting[i + 1];
                }
                waitingCount--;
                System.out.println("Barber cutting hair of customer " + cust);
            } finally {
                lock.unlock();
            }
            try {
                Thread.sleep(random.nextInt(3) + 1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("Barber finished with customer " + cust);
            lock.lock();
            try {
                condition.signal();
            } finally {
                lock.unlock();
            }
        }
    }
```

```
public void customer(int i) {
        try {
            Thread.sleep(random.nextInt(4) + 1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        lock.lock();
        try {
            if (waitingCount < NUM_CHAIRS) {
                waiting[waitingCount++] = i;
                System.out.println("Customer " + i + " waiting");
                condition.signal();
                while (waiting[0] != i) {
                    try {
                        condition.await();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                System.out.println("Customer " + i + " got haircut");
            } else {
                System.out.println("Customer " + i + " left (no seat)");
            }
        } finally {
            lock.unlock();
        }
    }
```

34

Barber in Java

```
public static void main(String[] args) {
        BarberShop barberShop = new BarberShop();
        Thread barberThread = new Thread(() -> barberShop.barber());
        barberThread.start();
        for (int i = 0; i < 5; ++i) {
            new Thread(() -> barberShop.customer(i)).start();
        }
        try {
            barberThread.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

35

Condition Variables

36

Condition Variables

- A Condition Variable allow the synchronization between threads that depend on the Condition Variable boolean value
- It allows a thread to wait for a condition that depends on the actions of another thread
  - The variable condition is a boolean value that depends on shared variables
- The Condition Variables is always associated to a latch:
  - Latch that protects the critical sections that control the access to the on shared variables on which the Condition Variable depends on
  - There can be more than one single Condition Variable associated to the same latch

37

Condition Variables - operations

- **await() (or wait()):**
  - A thread calls this method to suspend its execution and release the associated lock atomically, placing itself in a wait queue until it is signalled by another thread.
  - When the thread wakes up, it re-acquires/tries to lock before continuing.
- **signal() (or notify()):**
  - A thread calls this method after changing the shared state to wake up a single thread that is waiting on the condition variable.
  - If there are threads waiting (who have invoked wait), unblocks one thread (that goes to the state "to run")
  - The thread that was unblocked can not progress if it does not have the latch
  - Signal is not memorized; so, if there are no threads waiting in the condition variable, the signal has no effect
- **signalAll() (or notifyAll()):**
  - This method wakes up all threads waiting on the condition variable
  - This is generally safer and more foolproof than signal() in complex scenarios.
- **Multiple Conditions Per Lock**:
  - A single Lock instance can have multiple Condition objects associated with it, allowing threads to wait for different, specific conditions (e.g., notFull and notEmpty in a bounded buffer).
  - The Object.wait() mechanism only has one implicit wait set per object.
- **Flexibility and Readability**:
  - The explicit Lock and Condition objects make the synchronization logic clearer and more structured, reducing the chance of errors.
- **Timed Waits**:
  - The Condition interface provides methods like await(long time, TimeUnit unit) for timed waits, which can help prevent deadlocks or handle cases where a condition never occurs.
- **wait:**
  - Wait should be invoked while the latch is closed
- **signal/broadcast:**
  - After a signal has been received by the threads (that were blocked) the latch is not automatically acquired by the same thread (Mesa)
  - It is up to the programmer to posteriorly release the latch (Mesa)

38

Typical Examples

Code that waits for a Condition

```
lock(latch);
/* ..acess to shared variables.. */
while (! conditionOverTheSharedState()) wait(ConditionVar, latch);
/* ..access to shared variables.. */
unlock(latch);
```

Code that activates the Condition

```
lock(latch);
/* ..acess to shared variables.. */

/* if the state has been modified in a way that allows other
threads to progress, it calls either signal (or broadcast)*/
signal/broadcast (ConditionVar, latch);
unlock(latch);
```

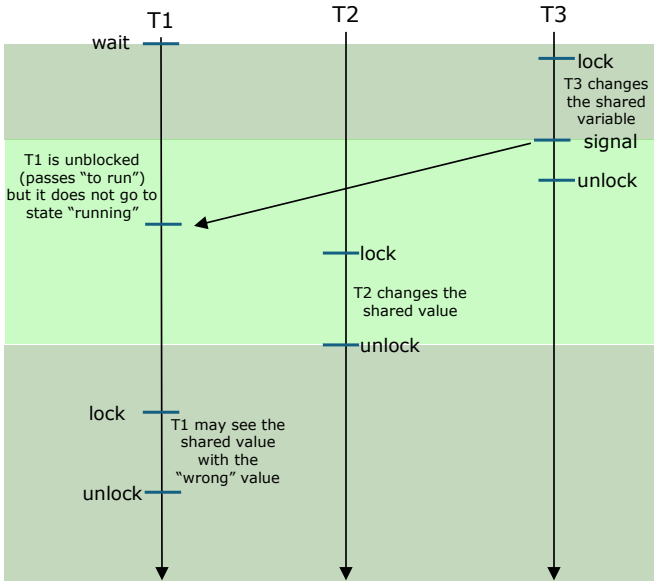
It is crucial to release the lock in a `finally` block to prevent the thread from holding the lock indefinitely in case of an exception. Also, notice the use of a `while` loop when calling `await()` (or `wait()`) to handle spurious wakeups, which can occur even if no signal was sent.

39

Condition Variables - problem

- Attention:
  - During the time that passes between the signal (T3) and a thread be unblocked (T1) acquiring the latch
  - The condition variable can be changed by another thread (T2)

Semantics Mesa (here is assumed) vs Semantics Hoare (see monitors)



40

Condition Variables - solution

- When a thread is unblocked (returns from wait):
  - It is not guaranteed that the value of the shared variables allow the thread to continue:
    - in the previous slide: the thread that is unblocked (T1) can not be the first to enter into the critical section after the thread that has performed the signal (T3)
  - **It should not be done:** if (test shared variable) wait
  - **It should be done**: while (test shared variable) wait

- Errors to avoid:
  - Do not close the latch before (outside the critical section) invoking wait: it can make the thread to not wait
  - Do not open the latch after doing signal (outside the critical section) : it can not unblock the thread waiting

41

“If vs While” in Condition Variables

- Always use `WHILE` instead of `IF`
- When you check a Condition Variable, using `while` instead of `if` is always correct
- The correction of using `if` depends on the semantics of signal
- The use of `while` also contemplates the situations in which occur spurious wakeups:
  - In some implementations, it is possible that two threads be unblocked in spite of being done just one signal

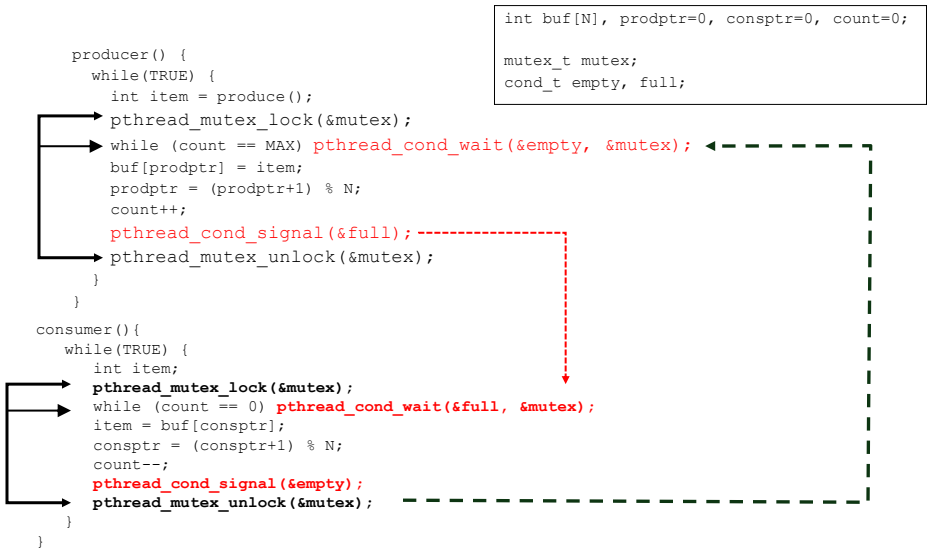
42

Condition Variables - POSIX

- Creation/destruction of Condition Variables:
  - pthread_cond_init (condition,attr)
  - pthread_cond_destroy (condition)
  - pthread_condattr_init (attr)
  - pthread_condattr_destroy (attr)
- Acquire and release in Condition Variables:
  - pthread_cond_wait (condition,mutex)
  - pthread_cond_signal (condition)
  - pthread_cond_broadcast (condition)

43

Producers – Consumers with Condition Variables



44

End of L20v26

57