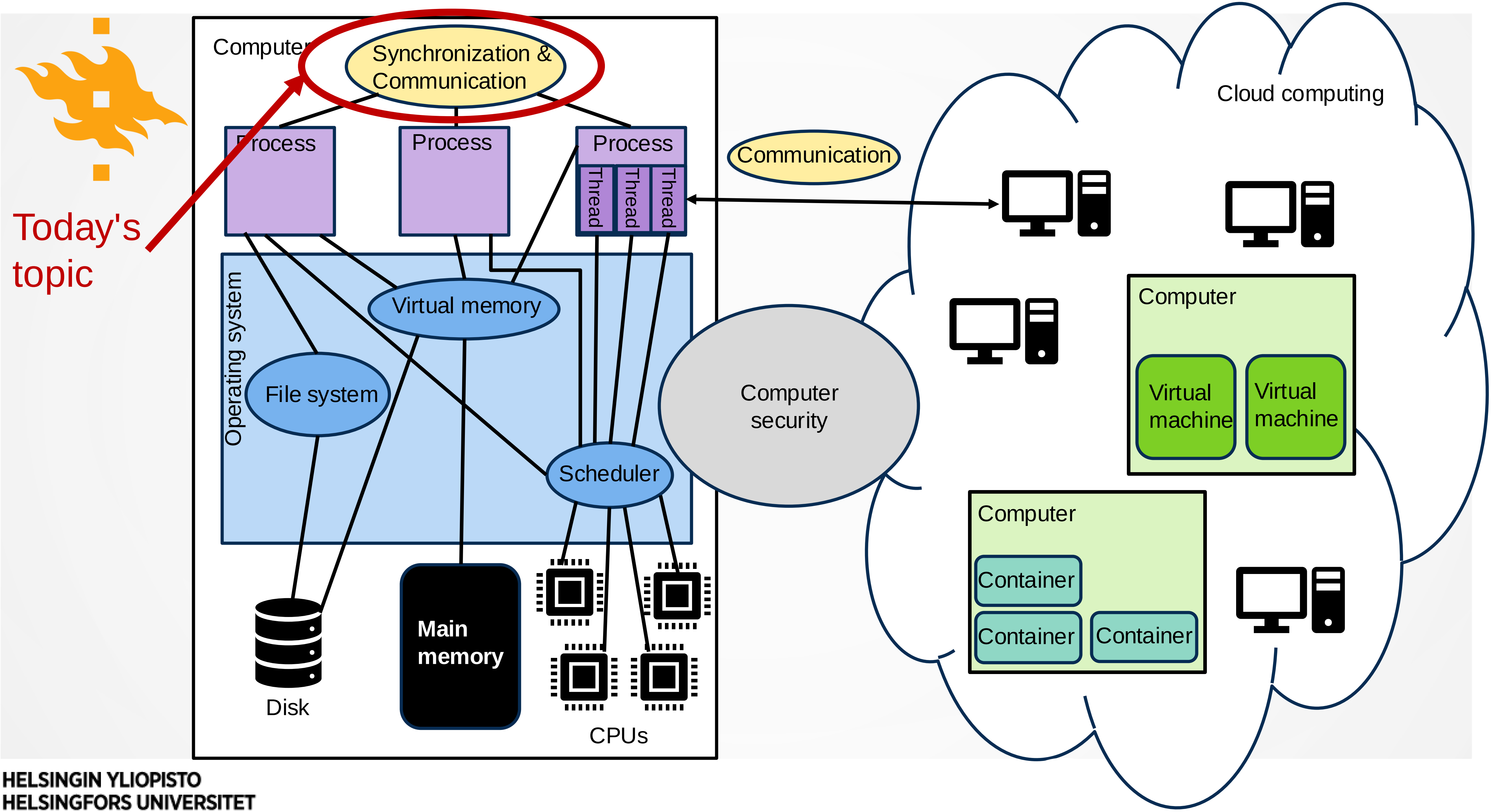
COMPUTING PLATFORMS

Concurrency: Basic concepts and locks and condition variables


Today's topic
Computer
Synchronization & Communication
Process
Process
Process
Thread
Thread
Thread
Communication
Operating system
Virtual memory
File system
Scheduler
Computer security
Disk
Main memory
CPUs
Cloud computing
Computer
Virtual machine
Virtual machine
Computer
Container
Container
Container

LEARNING OUTCOMES

- After today's lecture you
  - Know basic concepts related to concurrency such as race conditions and mutual exclusion requirements
  - Understand hardware approaches to supporting mutual exclusion
  - Are able to apply locks and condition variables to simple concurrency problems
  - Can describe and explain producer/consumer problem

EXAMPLE: RACE CONDITION

- Consider the code on the right:
  - X is a variable accessed by two threads (left and right column).
  - The two threads are run concurrently.
  - Single machine instructions can be considered **atomic operations** (no other process can see an intermediate state or interrupt the operation).
  - What are the possible values of X after both threads have executed?

```
shared int X;

load r1=5                load r1=11
store r1, X              store r1, X
```

EXAMPLE: ANOTHER RACE CONDITION

- Consider the code on the right:
  - X is a variable accessed by two threads (left and right column).
  - The two threads are run concurrently.
  - Single machine instructions can be considered **atomic operations.**
  - What are the possible values of X after both threads have executed?

```
shared int X = 0;

int i = 1;                     int j = 1;
while(i <= 2) {                while(j <= 2) {
   X = X + 1;                     X = X + 1;
   i = i + 1;                     j = j + 1;
}                              }
```

EXAMPLE: ANOTHER RACE CONDITION

X	Thread1	Thread2
0	Load X	
1	Write X+1	
1	Load X	
2	Write X+1	
2		Load X
3		Write X+1
3		Load X
4		Write X+1

X	Thread1	Thread2
0	Load X	
0		Load X
1	Write X+1	
1	Load X	
2	Write X+1	
1		Write X+1
1		Load X
2		Write X+1

X	Thread1	Thread2
0	Load X	
0		Load X
1		Write X+1
1	Write X+1	
1	Load X	
2	Write X+1	
2		Load X
3		Write X+1

…

CONTEXTS OF CONCURRENCY

- Multiple applications
  - **Multiprogramming** invented to allow processing time to be dynamically shared among several active applications
- Structured applications
  - Extension of modular design and structured programming: applications have concurrent processes/threads
- Operating system structure
  - OS implemented as a set of processes/threads

PRINCIPLES OF CONCURRENCY

- **Interleaving** (execution of processes/threads interleaved on a single processor) and **overlapping** (execution of processes/threads happens concurrently on multiple processors)
  - Can be viewed as examples of concurrent processing
  - Both present the same problems – concurrency
- Uniprocessor: relative speed of processes/threads cannot be predicted, depends on
  - Activities of other processes
  - How OS handles interrupts, and
  - Scheduling policies of OS

WHAT IS DIFFICULT TO HANDLE CONCURRENTLY?

- Sharing global resources
- Manage resources optimally
  - No knowledge of who is competing for access
- Difficult to locate programming errors as results are **nondeterministic** and **not reproducible**
  - Errors may only manifest in a rare sequence of events
  - Very hard to reproduce

PROBLEMS COMMONLY OCCURRING IN CONCURRENT PROCESSING

- **Mutual exclusion:**
  - When one process is in a **critical section** accessing shared resources, **no other process can be in a critical section** accessing the same shared resources
- **Deadlock:**
  - Two or more processes are unable to proceed because they are waiting for one of the others to do something
- **Starvation:**
  - A **runnable process is overlooked indefinitely by the scheduler**; although it is able to proceed, it is never chosen

CORRECTNESS OF CONCURRENCY PROBLEM SOLUTIONS

- How do you know that a solution does not work correctly?
  - Find one scenario where the solution does not work (produce a **counter example**)
- How do you know that a solution work correctly?
  - It produces correct result (always, in every scenario)
  - There is no deadlock
  - There is no starvation
  - Prove it: Use formal methods to mathematically prove your point (can be difficult; produces a **proof** of correctness)

CRITICAL SECTIONS AND MUTUAL EXCLUSION

- **Critical section**:
  - Section of code in a process that requires access to shared resources, and that **must not be executed while another process is in a corresponding section of code**
- **Mutex** (=mutual exclusion) solution:
  - Preprotocol: wait is needed, how?
  - Critical section
  - Postprotocol: wake up waiting process

```
/*Process 1*/
void P1
{
  while (true) {
    /*preceding code*/
    enter_critical(res
    /*critical section
    exit_critical(reso
    /*following code*/
  }
}
```

```
/*Process 2*/
void P2
{
  while (true) {
    /*preceding code*/
    enter_critical(resource)
    /*critical section*/
    exit_critical(resource)
    /*following code*/
  }
}
```

...

REQUIREMENTS FOR MUTEX SOLUTION

- Mutex must be enforced (functionally correct)
- **Fairness:** Does any thread get a fair chance of entering the critical section?
- **No starvation**
- **No deadlock**
- **Performance**:
  - If there is no contention (i.e. only one thread trying to access the critical section), what is the overhead of acquiring or releasing the mutex?
  - If there are multiple threads on a single CPU trying to access the critical section, are there performance concerns?
  - What about multiple threads on multiple CPUs?

MUTEX: DISABLING INTERRUPTS

- Works only on uniprocessor systems
- Guarantees mutual exclusion
- Waiting happens in the scheduler's Ready-queue
- Disadvantages:
  - Allowing user processes to control interrupts. Can be abused!
  - Efficiency of execution could be noticeably degraded.
  - Does not work in multiprocessor systems.

```
void lock() {
  disable_interrupts();
}

void unlock() {
  enable_interrupts();
}
```

MUTEX: A FAILED ATTEMPT

- The code on the right does not work correctly. Why?

```
int mutex_flag = 0;

void lock() {
  while (mutex_flag == 1){
  }
  mutex_flag = 1;
}

void unlock() {
  mutex_flag = 0;
}
```

MUTEX: A FAILED ATTEMPT

- The code on the right does not work correctly. Why?
- In `lock()` checking `mutex_flag` and setting it not atomic
- Consider two threads executing `lock()` concurrently:
  - Thread 1 checks that `mutex_flag` is 0
  - Thread 2 checks that `mutex_flag` is 0
  - Thread 1 sets `mutex_flag` to 1
  - Thread 2 sets `mutex_flag` to 1
  - Thus both will enter the critical section at the same time!

```
int mutex_flag = 0;

void lock() {
  while (mutex_flag == 1){
  }
  mutex_flag = 1;
}

void unlock() {
  mutex_flag = 0;
}
```

MUTEX: SPECIAL HARDWARE INSTRUCTIONS

- Compare-and-swap instruction (CAS)
  - Comparison is made between a memory value and a test value
  - If the values are the same, a swap occurs
  - Returns the old memory value
  - Carried out atomically

```
int compare_and_swap(int *word, int testval, int newval) {
  int oldval = *word;
  if (oldval == testval) {
    *word = newval;
  }
  return oldval;
}
```

MUTEX: COMPARE-AND-SWAP

```
int mutex_flag = 0;     /*0=free, 1=locked*/

void lock() {
  while(compare_and_swap(&mutex_flag, 0, 1) == 1) {
    /*do nothing*/
  }
}

void unlock() {
  mutex_flag = 0;
}
```

```
int compare_and_swap(int *word, int testval, int newval) {
  int oldval = *word;
  if (oldval == testval) {
    *word = newval;
  }
  return oldval;
}
```

MUTEX: COMPARE-AND-SWAP

```
int mutex_flag = 0;    /*0=free, 1=locked*/

void lock() {
  while(compare_and_swap(&mutex_flag, 0, 1) == 1) {
    /*do nothing*/
  }
}

void unlock() {
  mutex_flag = 0;
}
```

Atomic machine instruction, returns original value

Locked originally

Set locked (newval)

If open

```
int compare_and_swap(int *word, int testval, int newval) {
  int oldval = *word;
  if (oldval == testval) {
    *word = newval;
  }
  return oldval;
}
```

MUTEX: COMPARE-AND-SWAP

- Advantages
  - Applicable to any number of processes on either uniprocessor or multiprocessor environments
  - Simple and easy to verify
  - Can be used to support multiple critical sections, each with its own variable
- Disadvantages
  - Busy-waiting is employed. Thus a process consumes processor time while waiting
  - Starvation is possible when a process leaves critical section and more than one process is waiting (no FIFO queue)
  - Deadlock is possible

SPINLOCKS

- Compare-and-swap mutex implements a **spinlock**
  - Problem: busy-waiting (process is in a loop waiting for the lock to be released)
  - Spinlocks should only be used for very short critical sections (to minimize busy-waiting time)
- OS can use a spinlock to implement a lock without busy waiting
  - Use a spinlock to protect a critical section that sets the status of the lock (free/locked) and puts threads/processes into Blocked state to wait for the lock to be released
  - When the lock is released, a thread/process can be moved from the Blocked state to the Ready state

LOCKS AND CONDITION VARIABLES

- To implement concurrent programs, we need to solve two problems
  - **Critical section**: At most one thread/process can be inside a critical section at a time
  - **Synchronization**: A thread/process needs to wait for some condition to become true (e.g. data becoming available in a buffer)
- **Locks** solve the critical section problem and **condition variables** solve the synchronization problem
- Many programming languages implement locks and condition variables in a single construct called a **monitor**
  - E.g. Concurrent Pascal, Pascal-Plus, Modula-2, Modula-3, Java, Python (via threading.Condition object)
  - Software module consisting of one or more procedures, an initialization sequence, and local data

CRITICAL SECTIONS WITH LOCKS

- Two functions:
  - **Acquire**, lock
  - **Release**, unlock
- A critical section is implemented by acquiring the lock before the critical section and releasing it afterwards

```
lock = threading.lock()

lock.acquire()
# Critical section
lock.release()
```

SYNCHRONIZATION WITH CONDITION VARIABLES

- Condition variables have no value.
- A condition variable is associated with a FIFO queue of waiting processes
- Condition variables are operated with two functions:
  - **Wait**: suspend the calling process on the given condition
  - **Notify**: resume execution of one process blocked on the given condition (if any)
- Condition variables always associated with a lock to protect the internal state of the condition variable

```
lock = threading.lock()
condition = threading.condition(lock)

# Waiting thread                                    # Notifying thread
condition.acquire()                                 condition.acquire()
condition.wait()                                    condition.notify()
condition.release()                                 condition.release()
```

CONDITION VARIABLES: WAIT

- Calling process **always suspends**, process placed in queue
- **Unlocks the lock** associated with the condition variable
  - Allows someone else in the critical section
  - Allows another process awakened from (another?) wait to proceed
- When awakened, the process waits for the lock to proceed
  - May not be the next process to acquire the lock!
  - Condition waited for may not be true anymore, when the process acquires the lock and continues execution – **need to recheck the condition**!

CONDITION VARIABLES: NOTIFY

- Wakes up **first waiting process** if there are any
- Process continues execution, i.e. **it does not release the lock** (Lampson-Redell semantics)
- If no process is waiting in the queue, the notification is lost (no memory)
  - Advanced notifying (with memory) must be handled in some other way (use local variables protected by the lock associated with the condition variable)

- Alternative signaling semantics exist: Signaling process releases the lock and the signaled process continues execution in critical section (Hoare semantics)
  - Most practical implementations use Lampson-Redell semantics

EXAMPLE: PARENT WAITING FOR CHILDREN

- Parent thread creates one or more child threads
- Parent thread wishes to wait for the child thread(s) to finish
  - The parent should sleep until the child threads are done
  - When a child finishes, it should notify the parent thread

```
import threading

def child():
    print("child")
    # notify parent that I am done
```

```
def parent():
    print("parent start")
    child1 = threading.Thread(target=child)
    child2 = threading.Thread(target=child)
    child1.start()
    child2.start()
    # wait for children to be done
    print("parent end")
```

EXAMPLE: PARENT WAITING FOR CHILDREN

```
import threading

count = 0
lock = threading.Lock()
done = threading.Condition(lock)

def child():
    global count
    print("child")
    done.acquire()
    count -= 1
    if count == 0:
        done.notify()
    done.release()
```

```
def parent():
    global count
    count = 2
    print("parent start")
    child1 = threading.Thread(target=child)
    child2 = threading.Thread(target=child)
    child1.start()
    child2.start()
    done.acquire()
    while count != 0:
        done.wait()
    done.release()
    print("parent end")
```

PRODUCER/CONSUMER PROBLEM

- General situation:
  - One (or more) producer is generating data items and placing them in a buffer (one at a time)
  - One (or more) consumer is taking items out of the buffer (one at a time)
  - Producers and consumers may access the buffer concurrently
- Problem: How to ensure that
  - … producer must wait for consumer if buffer is full, and
  - … consumer must wait for producer if buffer is empty
- Who communicates with whom? How? (Communication problem)
- Who waits for whom? When? How? (Synchronization problem)

PRODUCER/CONSUMER: INFINITE BUFFER

- First, assume infinite buffer (cannot overflow!)
  - Producer does not need to check if there are free slots in the buffer
- We will implement this with a Python list where we always can append a new element

PRODUCER/CONSUMER PROBLEM: INFINITE BUFFER

```
import threading

buffer = []
lock = threading.Lock()
notempty = threading.Condition(lock)

def producer():
    global buffer
    while(True):
        item = produce_item()
        lock.acquire()
        buffer.append(item)
        notempty.notify()
        lock.release()
```

```
def consumer():
    global buffer
    while(True):
        lock.acquire()
        while len(buffer) == 0:
            notempty.wait()
        item = buffer.pop(0)
        lock.release()
        consume_item(item)
```

PRODUCER/CONSUMER PROBLEM: INFINITE BUFFER

```
import threading

buffer = []
lock = threading.Lock()
notempty = threading.Condition(lock)

def producer():
    global buffer
    while(True):
        item = produce_item()
        lock.acquire()
        buffer.append(item)
        notempty.notify()
        lock.release()
```

```
def consumer():
    global buffer
    while(True):
        lock.acquire()
        while len(buffer) == 0:
            notempty.wait()
        item = buffer.pop(0)
        lock.release()
        consume_item(item)
```

Protect critical section

Wait until buffer not empty
Why use while, not if?

Notify: there are now items in the buffer

PRODUCER/CONSUMER: FINITE BUFFER

- Block:
  - Insert in full buffer (producer waits)
  - Remove from empty buffer (consumer waits)
- Unblock
  - Item inserted (producer notifies consumer)
  - Item removed (consumer notifies producer)

PRODUCER/CONSUMER PROBLEM: FINITE BUFFER

```
import threading

buffer = []
lock = threading.Lock()
notempty = threading.Condition(lock)
notfull = threading.Condition(lock)

def producer():
    global buffer
    while(True):
        item = produce_item()
        lock.acquire()
        while len(buffer) == 10:
            notfull.wait()
        buffer.append(item)
        notempty.notify()
        lock.release()
```

```
def consumer():
    global buffer
    while(True):
        lock.acquire()
        while len(buffer) == 0:
            notempty.wait()
        item = buffer.pop(0)
        notfull.notify()
        lock.release()
        consume_item(item)
```

PRODUCER/CONSUMER PROBLEM: FINITE BUFFER

```
import threading

buffer = []
lock = threading.Lock()
notempty = threading.Condition(lock)
notfull = threading.Condition(lock)

def producer():
    global buffer
    while(True):
        item = produce_item()
        lock.acquire()
        while len(buffer) == 10:
            notfull.wait()
        buffer.append(item)
        notempty.notify()
        lock.release()
```

```
def consumer():
    global buffer
    while(True):
        lock.acquire()
        while len(buffer) == 0:
            notempty.wait()
        item = buffer.pop(0)
        notfull.notify()
        lock.release()
        consume_item(item)
```

Wait until buffer not full
Why use while, not if?

Notify: the buffer is not full

Wait until buffer not empty
Why use while, not if?

Notify: there are now items in the buffer

SUMMARY

- Critical sections and mutual exclusion (mutex)
- Spinlocks with hardware assistance
- Locks and condition variables
  - Locks implement mutual exclusion
  - Condition variables implement synchronization
- Producer/consumer problem