

Operating Systems: Synchronization

CSC-4320/6320 –Summer 2014

Synchronization

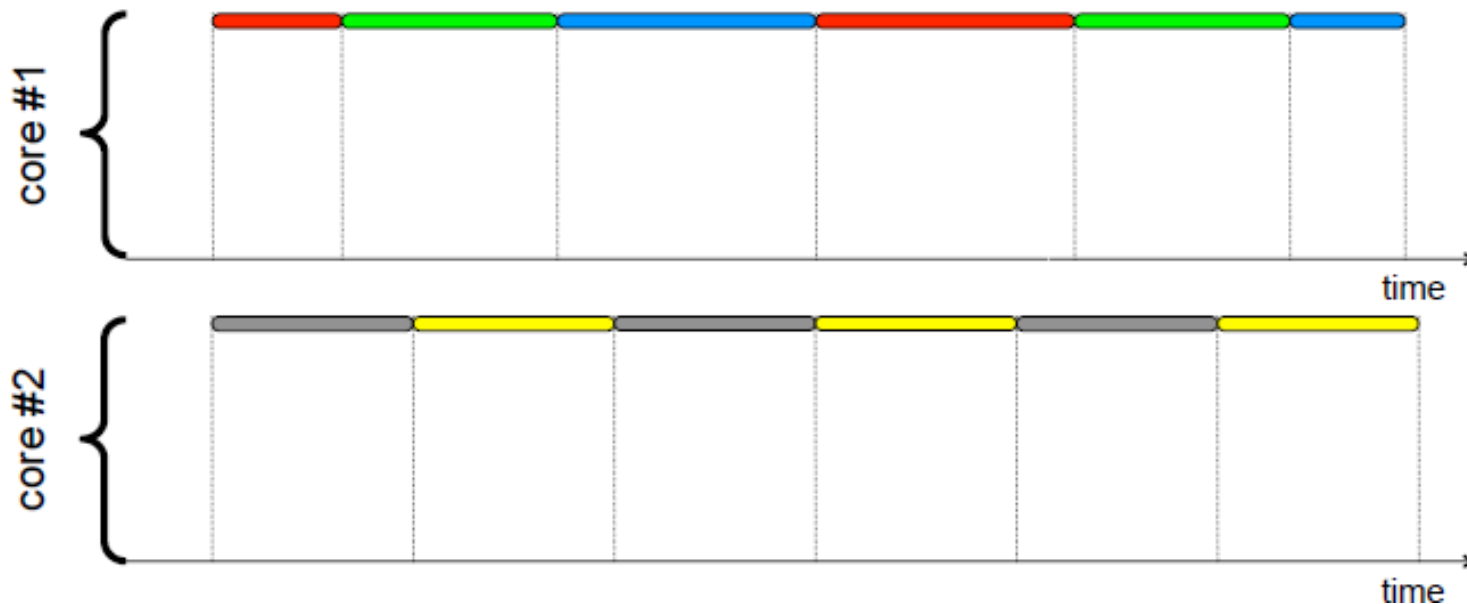
- Synchronization is an important topic and it is difficult to do justice in just a few lectures in an OS course
 - Though that is often done
- We are going to cover the most important parts in this class
- There is much more to learn than we will cover and you should pursue a more in-depth study

Cooperating Processes/Threads

- Having execution units run **concurrently** is useful
 - Structuring an application as independent but cooperating entities can be very convenient
 - Better utilization of hardware resources (e.g., cores)
- Ways of doing concurrency
 - Multiple processes and message-passing
 - or shared memory segments, but less fashionable
 - Multiple threads in a single address space
 - All of the above together!

Concurrency

- Two kinds of concurrency:



- false concurrency within a core: illusion of concurrency provided by the OS (e.g. green and blue tasks)
- true concurrency across cores (e.g., green and yellow task)

True/False Concurrency

- The programmer shouldn't have to care/know whether concurrency will be true or false
 - Typically, the programmer doesn't know on which core the program will run in the end!
- A concurrent program with 10 tasks should work on a single-core processor, a quad-core, a 32-core processor, etc.
- However, better performance with true concurrency
- We have talked about true concurrency across cores, but there could be true concurrency between any two hardware resources
 - e.g., between the network card and the core
 - e.g., between the disk and the network card

Concurrency Dangers

- There are two main problems with concurrent programs:
 - **Race Conditions**: a bug that leads the program to give unpredictably incorrect results
 - Typical with processes/threads sharing memory
 - **Deadlocks**: the program blocks forever
 - Possible in any distributed system
- Let's first talk about Race Conditions
 - Arguably the most common/vexing problems
 - You will, unfortunately, encounter them
 - Deadlocks are in their own lecture

Race Condition

- Consider a multi-threaded program in which we have
 - A counter (in the data segment)
 - A thread that increments the counter for n iterations
 - A thread that decrements the counter for n iterations
- No, it is not a useful program
- Question: what is the final value of the counter?

Why Race Conditions?

- Race conditions can happen with false or true concurrency
 - Everything else being equal, one could argue that they're most statistically likely to manifest themselves with true concurrency
- The count += 1 and count -= 1 statements are written in a high-level language
- The compiler translates them into machine code (or byte code if we are talking Java) which we like to look at as assembly code
- On a Load/Store architecture (RISC), the code would then be:

```
; Thread #1  
load    R1, [@]  
inc     R1  
store   [@], R1
```

```
; Thread #2  
load    R1, [@]  
dec     R1  
store   [@], R1|
```


Why Race Conditions?

- Illusion of concurrency: the OS context-switches threads rapidly
- We have 2 sets of 3 instructions, and thus many possibilities
- Three possible execution paths

load	R1, [@]
inc	R1
load	R1, [@]
dec	R1
store	[@], R1
store	[@], R1

load	R1, [@]
inc	R1
load	R1, [@]
dec	R1
store	[@], R1
store	[@], R1

load	R1, [@]
load	R1, [@]
dec	R1
inc	R1
store	[@], R1
store	[@], R1

Important: **R1** is not the same as **R1**

They are both register values into logical register sets (i.e., inside a data structure in the OS)

Why Race Conditions?

Let's assume that initially $[@] = 5$

```
load  R1, [@] // R1 = 5
inc    R1      // R1 = 6
load  R1, [@] // R1 = 5
dec    R1      // R1 = 4
store [@], R1  // [@] = 4
store [@], R1  // [@] = 6
```

```
load  R1, [@] // R1 = 5
load  R1, [@] // R1 = 5
dec    R1      // R1 = 4
inc    R1      // R1 = 6
store [@], R1  // [@] = 4
store [@], R1  // [@] = 6
```

```
load  R1, [@] // R1 = 5
inc    R1      // R1 = 6
load  R1, [@] // R1 = 5
dec    R1      // R1 = 4
store [@], R1  // [@] = 6
store [@], R1  // [@] = 4
```

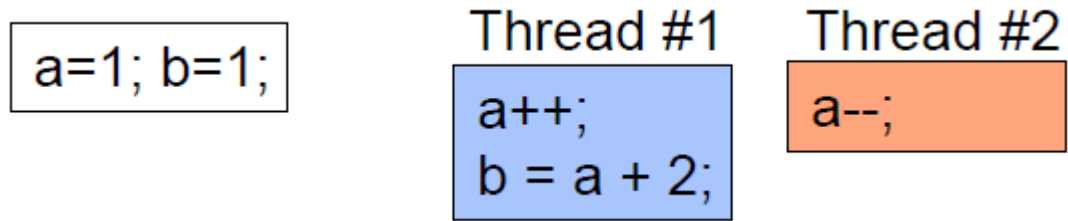
We would expect $[@]$ to be 5 at the end
But we get 4 or 6

Lost Update

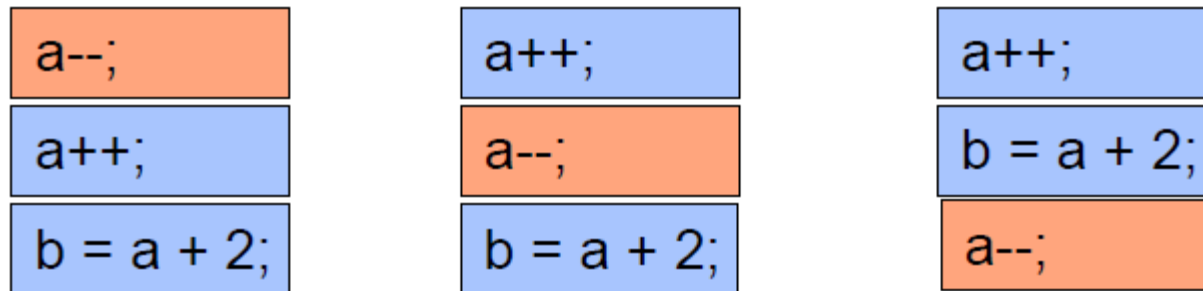
- In general, when a thread does “x++” and another does “+--” three things can happen
 - Both updates go through, the x is unchanged
 - The “x++” update is lost, and the value of x is decremented only
 - The “x--” update is lost, and the value of x is incremented only

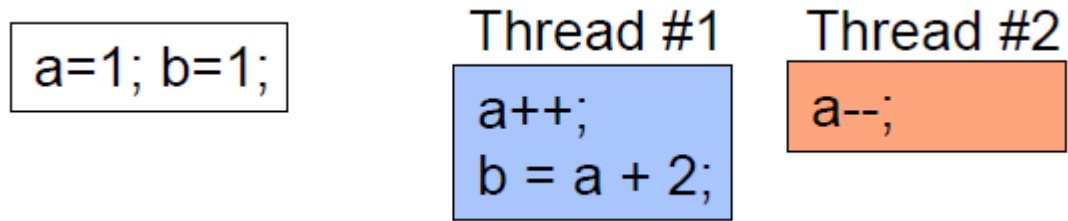
Race Condition Example

- Assume we have two global variables a and b, initially both set to 1
- Thread #1:
 a++;
 b=a+2;
- Thread #2:
 a--;
- Once both threads are finished, the main thread prints the value of a and b
- Question: what are the possible values?

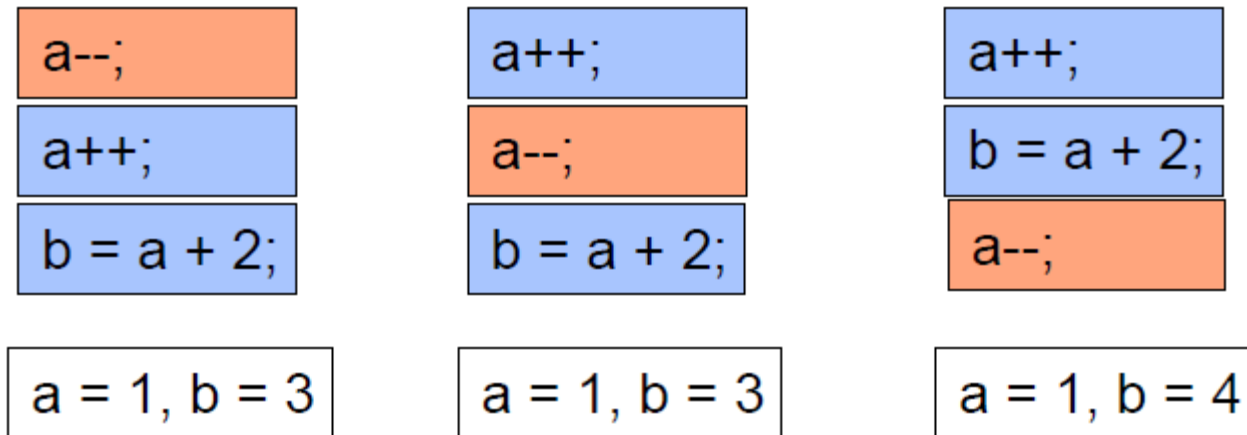


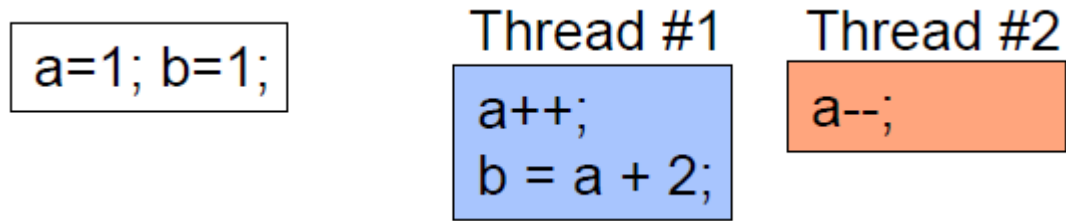
- First thing to do: come up with all possible interleaving of the instructions assuming that all instructions execute entirely without being interrupted



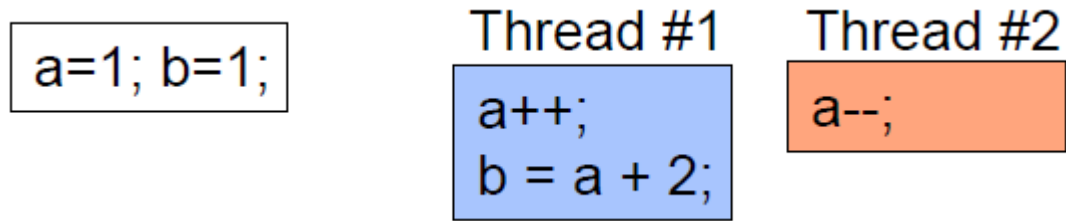


- First thing to do: come up with all possible interleaving of the instructions assuming that all instructions execute entirely without being interrupted





- Second thing to do: lost updates
 - Each line of code consists of multiple “hardware” instructions
- In this case: bad interaction between “a++” and “a--”
 - Result: a=2
 - “a--” reads value 1, computes 0, gets interrupted
 - “a++” reads value 1, computes 2, get interrupted
 - “a--” writes value 0
 - “a++” writes value 2, overwriting the 0
 - Result: a = 0
 - Same as “a=2” just different order
 - Result: a = 1
 - Everything went well, without lost update
 - We end up with two new possible output: a = 0, b = 2 a = 2, b = 4



- Output produced for all possible interleaving of lines of code
 - Can be considered a bug or not depending on what your application does
 - An application must not necessarily be 100% deterministic to be correct
 - Input could be random anyway
- Output produced due to the lost update problem
 - Typically considered a bug because a has a value different from 1 after “a++” and “a--” in the code, and b can take value 2 which likely makes no sense

a = 1, b = 3

a = 1, b = 3

a = 1, b = 4

a = 0, b = 2


a = 2, b = 4

Why do we Hate Race Conditions?

- A code may be working fine a million times, and then fail one day, and then it takes you one million times to reproduce the bug
- If you modify the code (e.g., adding a few print statements), or if you run in debugging mode, the race condition may no longer manifest itself or manifest itself more
 - The famous “I just added a print and everything works!”
- If you write code, run it, and it works, you don’t really know whether you’ve written a bug-free program
 - Typically true, but exacerbated with race conditions
 - You can prove a program wrong, but not a program right!
- **We hate nondeterministic bugs**
 - and we hate bugs to begin with...
- **So what do we do?**

Critical Section

- We want a **critical section**: a section of the code in which **only one thread** can be at a time
 - It doesn't have to be a contiguous section of code
 - In the example here, we have a 3-zone critical section
- If thread A is already in one of the “orange zones”, then all other threads are blocked before being allowed to enter any orange zone
 - And only one will be allowed to enter once thread A leaves the orange zone it was in



```
Terminal — vim — 101x71

/*
 * print_queue()
 */
void print_queue(xbt_fifo_t q) {
    xbt_fifo_item_t item;
    job_descriptor_t descriptor;

    fprintf(stderr, "----> %d", xbt_fifo_size(q));
    xbt_fifo_foreach(q, item, descriptor, job_descriptor_t) {
        fprintf(stderr, " [%d, %d, %d, %d] ", descriptor->id, descriptor->nodes,
            descriptor->requested_duration, descriptor->actual_duration);
    }
    fprintf(stderr, "\n");
}

/*
 * start_job()
 */
void start_job(job_descriptor_t jd, scheduler_bookkeeping_st *bk)
{
    jd->start_time = MSG_get_clock();

    /* Notify the jobsimulator of a new job */
    {
        m_task_t task;
        task = MSG_task_create("job_started", 0, 0, (void*)jd);
        if (MSG_task_put(task, MSG_host_self(), PORT_START_JOB) != MSG_OK) {
            xbt_assert(0, "Error while sending a job start notification to the job simulator");
        }
    }

    /* Notify the submitter */
    {
        m_task_t task;
        int id = jd->id;
        task = MSG_task_create("job_started", 0, 0, (void*)id);
        if (MSG_task_put(task, jd->submitter, jd->channel_started) != MSG_OK) {
            xbt_assert(0, "Error while sending a job start notification");
        }
    }

    /* Notify the submitter of my queue size */
    /* (this could really be done anywhere) */
    {
        m_task_t task;
        int queue_size = xbt_fifo_size(bk->queued);
        task = MSG_task_create("queue_size", 0, 0, (void*)queue_size);
        if (MSG_task_put(task, jd->submitter, jd->channel_queue_size) != MSG_OK) {
            xbt_assert(0, "Error while sending my queue size");
        }
    }

    return;
}

/*
 * scheduler_init()
 */
void scheduler_init(job_scheduling_algorithm_t alg, scheduler_bookkeeping_st *bk)
{
    /* Initialize the job descriptor queues and the number of free nodes */
    bk->queued = xbt_fifo_new();
    bk->running = xbt_fifo_new();

    /* Alg-specific initialization */
    switch (alg) {
        case FCFS:
            scheduler_init_fcfs(bk);
    }
}
```

Critical Section

- We can have multiple critical sections
 - One 3-zone “orange” critical section
 - One 2-zone “green” critical section
- In our initial example, we’d simply put the count++ and count-- statements in a (possibly multi-zone) critical section



```
Terminal — vim — 101x71

/*
 * print_queue()
 */
void print_queue(xbt_fifo_t q) {
    xbt_fifo_item_t item;
    job_descriptor_t descriptor;

    fprintf(stderr, "----> %d", xbt_fifo_size(q));
    xbt_fifo_foreach(q, item, descriptor, job_descriptor_t) {
        fprintf(stderr, " [%d, %d, %d] ", descriptor->id, descriptor->nodes,
            descriptor->requested_duration, descriptor->actual_duration);
    }
    fprintf(stderr, "\n");
}

/*
 * start_job()
 */
void start_job(job_descriptor_t jd, scheduler_bookkeeping_st *bk)
{
    jd->start_time = MSG_get_clock();

    /* Notify the jobsimulator of a new job */
    {
        m_task_t task;
        task = MSG_task_create("job started", 0, 0, (void*)jd);
        if (MSG_task_put(task, MSG_host_self(), PORT_START_JOB) != MSG_OK) {
            xbt_assert(0, "Error while sending a job start notification to the job simulator");
        }
    }

    /* Notify the submitter */
    {
        m_task_t task;
        int id = jd->id;
        task = MSG_task_create("job started", 0, 0, (void*)id);
        if (MSG_task_put(task, jd->submitter, jd->channel_started) != MSG_OK) {
            xbt_assert(0, "Error while sending a job start notification");
        }
    }

    /* Notify the submitter of my queue size */
    /* (this could really be done anywhere) */
    {
        m_task_t task;
        int queue_size = xbt_fifo_size(bk->queued);
        task = MSG_task_create("queue size", 0, 0, (void*)queue_size);
        if (MSG_task_put(task, jd->submitter, jd->channel_queue_size) != MSG_OK) {
            xbt_assert(0, "Error while sending my queue size");
        }
    }

    return;
}

/*
 * scheduler_init()
 */
void scheduler_init(job_scheduling_algorithm_t alg, scheduler_bookkeeping_st *bk)
{
    /* Initialize the job descriptor queues and the number of free nodes */
    bk->queued = xbt_fifo_new();
    bk->running = xbt_fifo_new();

    /* Alg-specific initialization */
    switch (alg) {
        case FCFS:
            scheduler_init_fcfs(bk);
    }
}
```

Critical Section

- More formally, we want three properties of critical sections:
 - **Mutual exclusion**: if thread P is in the critical section, then no other thread can be in it
 - **Progress**: if thread P wants to enter into a critical section it will enter it eventually
 - **Bounded waiting**: once thread P has declared intent to enter the critical section, there is a bound on the number of threads that can enter the critical section before P
- Note that there is no assumption regarding the relative speed of the involved threads
 - But no thread has speed zero

Critical Section Misconception

- A Critical Section corresponds to sections of code (i.e., the text segment)
- It does not correspond to data (i.e., variables)
 - Even though the section of code is typically one that modifies a particular set of variables
- When we say “we need to protect variable x against race conditions” it means “we need to look at the entire code, see where x is modified, and put all those places in the SAME critical section”
 - If software engineering is well-done, modification of a single variable doesn’t happen all over the code
- It is a misconception that critical sections are attached to variables

Preemptive vs. Non-Preemptive

- A **preemptive** kernel allows a thread executing kernel code (in kernel mode) to be preempted
- A **non-preemptive** kernel does not
 - The thread runs until willingly exits kernel mode (or yields control of the CPU)
- Non-preemptive kernels are simple
 - There is no race condition (with single core CPUs)
- Preemptive kernels are more powerful
 - Better for real-time programming as a real-time thread can preempt a thread running in kernel mode
 - Should be more responsive for the same reason
- **Most modern kernels are preemptive**

Critical Sections and the Kernel

- On modern OSes, multiple threads can be in the kernel mode
 - User threads that are doing a system call and are in kernel mode
 - Threads started by the kernel itself to do useful kernel things
- Therefore, the kernel is subject to race conditions
 - We've seen that kernel debugging is hard, that race condition debugging is hard, so we don't want race conditions in the kernel
- Example: the kernel maintains many data structures
 - e.g., the list of open files
 - The list must be updated each time a file is opened or closed
 - This is very much like the counter++ / counter – example
 - e.g., the list of memory allocations
 - e.g., the list of processes
 - e.g., the list of interrupt handlers
- The Kernel developer must avoid all race conditions for accesses to these data structures

Synchronization Implementation

- What we need is a way to implement `enter_critical_section()` and `leave_critical_section()`
- There are some software “solutions”
 - They can be very complicated
 - They’re not guaranteed to work on modern architecture
 - See Section 6.3 in the book if interested
- What we need is help from the hardware
- One option: `disabling interrupts?`
- Problems:
 - If you allow whatever user process to disable interrupts, what tells you it will enable them afterwards?
 - What if interrupts are needed for other purposes, such as a bunch of timers?
- Conclusion: although inside the kernel one could disable interrupts for specific purposes, one cannot use this mechanism in general

Atomic Instructions and Locks

- Modern processors offer **atomic** instructions
 - Instructions uninterruptible from issue to completion
- With atomic instructions it is easy to implement the “lock” abstraction
- A lock is an abstract data type with two methods: **lock()** and **unlock()**
 - To “acquire” and “release” the lock
- A critical section is defined as the segments of code in between pairs of lock/unlock calls for a given lock
- Example

```
Lock mutex = new Lock();
```

```
...
```

```
mutex.lock();
```

```
//All code here is part of the critical section defined by mutex
```

```
mutex.unlock();
```

Short Critical Sections

- Critical sections should be as short as possible
 - Not in lines of code, but in time to run these lines
- Long critical sections: only one thread can do work for a while, so we have reduced parallelism
 - Extreme situation: the whole code is critical
 - Not a good idea in the case of multiple cores
- Goal: Many small and short critical sections (with different locks)
 - Many threads can do useful work simultaneously

What do Locks do?

- Two kinds of lock implementations
- **Spin lock:** The thread constantly checks whether the lock is available in a while loop
 - Prevents others (e.g., unrelated) threads from using CPU cycles
 - A big problem on a single-core system
 - Wastes power and dissipates heat
 - But the thread will acquire the lock “as soon as” it is released
 - Very little overhead, meaning no kernel involvement (obviously a huge waste of CPU time that could have been used by another thread doing useful work)
- **Blocking lock:** The thread asks the OS to be put in the Waiting/Blocked state and the OS will make the thread Ready whenever the lock has been released by another thread
 - Has higher overhead as system calls running kernel code, and locking/unlocking overhead is important
 - **But it does not waste CPU cycles by “spinning”!!**

Spin vs. Blocking Lock

- Spinlocks are very useful for (short) critical sections
 - Burn only a few cycles, but provide fast response time because they do not involve the kernel
 - If your critical section is “x++”, definitely use a spinlock, not a blocking lock
 - Spin locks are used inside the kernel for speed
- Most kernels provide a blocking lock abstraction as well
 - To be used for long(er) critical sections

Thread Synchronization?

- It may be tempting to use locks for having two threads communicate
 - Thread A waits for an “event” by doing `lock(x)`;
 - Thread B signals the “event” by doing `unlock(x)`;
- This is not a good idea, and a separate abstraction is needed
- This abstraction is called a **condition variable**
- It provides two mechanisms:
 - **`wait()`**: Ask the kernel to be put in the Blocked state
 - **`signal()`** and **`signal_all()`**: Unblock a (all) blocked thread(s)
 - i.e., tell the OS that the thread is runnable again
 - Does not mean that the thread calling `signal()` relinquishes the CPU immediately: it is only about some threads changing state
- Conceptually, the kernel has a queue of blocked threads for each condition variable

Thread #1

...

`cond.wait();`

...

Thread #2

...

`cond.signal();`

...

Cond Variables and Locks

- If a thread acquires a lock, and then calls wait() on a condition variable, then it is blocked and nobody else can get the lock!
 - General rule: do not go to sleep while you are holding a resource that could let a bunch of people do useful work (i.e., a lock)
- To enforce this, a condition variable is associated with a lock, and wait() temporarily releases the lock
 - This is safe because while a thread sleeps, it is not doing anything at all
- Pseudo-code for wait:

```
void wait(cond_t cond, lock_t mutex){
    unlock(mutex);
    <ask the OS to put me into the blocked state and to unblock me
        when the event “cond” is signaled>
    lock(mutex);
}
```

Classical Sync. Problems

- To explain/understand synchronization, many typical problems are used
- Some are things you'll implement often
 - Producer-Consumer, Reader-Writer, Bank account, ...
- Others are interesting metaphors
 - Dining philosophers, barber shop, ..
- Some are surprisingly difficult and finding good solutions has occupied many computer scientists
- We will only talk a little bit about Producer/Consumer

Producer/Consumer

- We have a bounded buffer
 - i.e., an array of fixed maximum size
- We have producer threads
 - Each producer runs forever and
 - At (random) times attempts to put an element in the buffer in the first empty slot
 - If there is no empty slot, the producer blocks (no busy waiting allowed)
 - A producer should never produce in a full buffer
- We have consumer threads
 - Each consumer runs forever and
 - At (random) times attempts to read an element from the first slot in the buffer
 - If there is no element to be read, the consumer blocks (no busy waiting allowed)
 - A consumer should never consume from an empty buffer
- In Section 3.4.1, the book had a bounded buffer example that was a little incomplete, and they revisit it in Chapter 6 with Semaphores
- Let's do it with locks and condition variables
 - Let's assume that we have a Lock and a CondVar class in Java

BoundedBuffer Class

```
public class BoundedBuffer<E> implements Buffer<E>{
    private final int BUFFER_SIZE=5;
    private E[] buffer;           //Array used to store the buffer
    private int in;               //Index of the next free slot
    private int out;              //Index of the next full slot
    private int count;            //Number of elements in the buffer

    public BoundedBuffer(){
        in=0; out=0; count=0;
        buffer = (E[])new Object[BUFFER_SIZE];
    }

    public void insert(E item){ //Called by producers
    }

    public E remove(){ //Called by consumers
    }
}
```

Objective:
implement insert()
and remove()

Terrible Implementation

```
- public void insert(E item) {  
    while(count==BUFFER_SIZE);  
    buffer[in]=item;  
    in= (in+1) % BUFFER_SIZE;  
    count++;  
}  
  
- public E remove() {  
    while(count==0);  
    E item = buffer[out];  
    out = (out+1) % BUFFER_SIZE;  
    count--;  
    return item;  
}
```

Terrible Implementation

```
public void insert(E item) {  
    while(count==BUFFER_SIZE);  
    buffer[in]=item;  
    in= (in+1) % BUFFER_SIZE;  
    count++;  
}  
  
public E remove() {  
    while(count==0);  
    E item = buffer[out];  
    out = (out+1) % BUFFER_SIZE;  
    count--;  
    return item;  
}
```

- Race conditions all over the place
- Busy waits
- Let's first deal with race conditions
- Which lines of code should be protected by mutual exclusion?

Terrible Implementation

```
- public void insert(E item) {  
    while(count==BUFFER_SIZE);  
    buffer[in]=item;  
    in= (in+1) % BUFFER_SIZE;  
    count++;  
}  
  
- public E remove() {  
    while(count==0);  
    E item = buffer[out];  
    out = (out+1) % BUFFER_SIZE;  
    count--;  
    return item;  
}
```

- Race conditions all over the place
- Busy waits
- Let's first deal with race conditions
- Which lines of code should be protected by mutual exclusion?

ALL

Terrible Implementation

```
public void insert(E item){  
    while(count==BUFFER_SIZE)  
        buffer[in]=item;  
    in= (in+1) % BUFFER_SIZE;  
    count++;  
}  
  
public E remove(){  
    while(count==0);  
    E item = buffer[out];  
    out = (out+1) % BUFFER_SIZE;  
    count--;  
    return item;  
}
```

Two producers could both see the count as BUFFER_SIZE-1, and proceed when there is space for only one item

Classical lost update among two producers

Two consumers could both see the count as 1, and proceed when there is only one item

Classical lost update among two consumers

Terrible Implementation

```
public void insert(E item) {  
    while(count==BUFFER_SIZE)  
        buffer[in]=item;  
    in= (in+1) % BUFFER_SIZE;  
    count++;  
}
```

Two producers could both see the count as BUFFER_SIZE-1, and proceed when there is space for only one item

Classical lost update among any threads

Classical lost update among two producers

```
public E remove() {  
    while(count==0);  
    E item = buffer[out];  
    out = (out+1) % BUFFER_SIZE;  
    count--;  
    return item;  
}
```

Two consumers could both see the count as 1, and proceed when there is only one item

Classical lost update among two consumers

Adding Mutual Exclusion

```
public class BoundedBuffer<E> implements Buffer<E>{  
    private final int BUFFER_SIZE=5;  
    private E[] buffer;           //Array used to store the buffer  
    private int in;               //Index of the next free slot  
    private int out;              //Index of the next full slot  
    private int count;            //Number of elements in the buffer  
    private Lock mutex;           //Lock  
  
    public BoundedBuffer(){  
        in=0; out=0; count=0;  
        buffer = (E[])new Object[BUFFER_SIZE];  
        mutex = new Lock();  
    }  
  
    public void insert(E item){ //Called by producers  
    }  
  
    public E remove(){ //Called by consumers  
    }  
}
```

The added
Lock

Adding Mutual Exclusion

```
public void insert(E item){  
    mutex.lock();  
    while(count==BUFFER_SIZE);  
    buffer[in]=item;  
    in= (in+1) % BUFFER_SIZE;  
    count++;  
    mutex.unlock();  
}  
  
public E remove(){  
    mutex.lock();  
    while(count==0);  
    E item = buffer[out];  
    out = (out+1) % BUFFER_SIZE;  
    count--;  
    mutex.unlock();  
    return item;  
}
```

- What's the problem?

Adding Mutual Exclusion

```
public void insert(E item){
    mutex.lock();
    while(count==BUFFER_SIZE);
    buffer[in]=item;
    in= (in+1) % BUFFER_SIZE;
    count++;
    mutex.unlock();
}

public E remove(){
    mutex.lock();
    while(count==0);
    E item = buffer[out];
    out = (out+1) % BUFFER_SIZE;
    count--;
    mutex.unlock();
    return item;
}
```

- What's the problem?
- **DEADLOCK!**
 - Consumer shows up first, acquires the lock, and then loops forever waiting for count !=0
 - Producer shows up next, tries to acquire the lock and blocks forever
 - Nothing happens, the program hangs

Adding Condition Variables

```
public class BoundedBuffer<E> implements Buffer<E>{
    private final int BUFFER_SIZE=5;
    private E[] buffer;           //Array used to store the buffer
    private int in;               //Index of the next free slot
    private int out;              //Index of the next full slot
    private int count;            //Number of elements in the buffer
    private Lock mutex;           //Lock
    private CondVar notFull;      //Condition variable (to signal "there is space!")
    private CondVar notEmpty;     //Condition variable (to signal "there is data!")

    public BoundedBuffer(){
        in=0; out=0; count=0;
        buffer = (E[])new Object[BUFFER_SIZE];
        mutex = new Lock();
        notFull = new CondVar();
        notEmpty = new CondVar();
    }
}
```

Adding Mutual Exclusion

```
public void insert(E item){
    mutex.lock();
    while(count==BUFFER_SIZE)
        notFull.wait(mutex); //Wait until there is space
    buffer[in]=item;
    in= (in+1) % BUFFER_SIZE;
    count++;
    notEmpty.signal(); //Tell (consumers) that there is something in the buffer
    mutex.unlock();
}

public E remove(){
    mutex.lock();
    while(count==0)
        notEmpty.wait(mutex); //Wait until there is data
    E item = buffer[out];
    out = (out+1) % BUFFER_SIZE;
    count--;
    notFull.signal(); //Tell (producers) that there is space in the buffer
    mutex.unlock();
    return item;
}
```

Use a Single Cond Variable

```
public void insert(E item){
    mutex.lock();
    while(count==BUFFER_SIZE)
        condVar.wait(mutex);
    buffer[in]=item;
    in= (in+1) % BUFFER_SIZE;
    count++;
    condVar.signalAll();
    mutex.unlock();
}

public E remove(){
    mutex.lock();
    while(count==0)
        condVar.wait(mutex);
    E item = buffer[out];
    out = (out+1) % BUFFER_SIZE;
    count--;
    condVar.signalAll();
    mutex.unlock();
    return item;
}
```

- Often developers take a brute-force approach:
- Use only one condition variable
- Wake up all threads
 - They all re-check the condition
 - And only one gets through because only one can re-acquire the lock
- This makes the code simpler, but causes performance hits
 - When you have 1000 threads, waking them all up to have 999 of them immediately block again leads to overhead

Monitors

- Writing concurrent programs with locks and condition variables is very error prone
 - Typically, either you are implementing a version of the well-known problems, or you are introducing concurrency bugs
 - At least as a beginner concurrent programmer
 - The producer-consumer wasn't super easy either
- In the 70s, Hoare/Brinch-Hansen proposed the concept of a **Monitor**
- A monitor is an abstract data type representing a shared “resource”
 - e.g., a class/object
- It is a construct of a programming language
- Java implements monitors
 - You can implement Lock and CondVar with Java monitors, but few people do this and just use monitors directly

Monitors

- There is nothing magical here, we still need the two basic functionalities of **mutual exclusion** and **waiting/signaling**
- Monitors have the same “power” as other synchronization abstractions such as locks and condition variables
- But monitors constrain several aspects
 - **Condition variables are not visible** outside the monitor
 - They are hidden/encapsulated
 - One interacts with them via special monitor operations
 - **Mutual exclusion is implicit**
 - Monitor operations execute by definition in mutual exclusion
- These apparently innocuous properties make writing concurrent code less error-prone
 - The programmer should not have to deal with lock, unlock, wait, and signal
- The book describes Monitors in Section 6.7 in detail

Synchronization in Java

- Unbeknownst to you, all Java objects you have used in your life have a lock and a condition variable “hidden” inside of them
 - And implement lock- and condvar-like methods
- To ensure mutual exclusion, a method can be declared as **synchronized**:
 - e.g., `public synchronized void addItem(Item E)`
- **All synchronized methods in a class are executed in mutual exclusion**
 - This is sometimes overkill or downright a hindrance, so one can also ensure mutual exclusion for a block of code or for a class, but Java isn't really the subject of this class, so you can look into it
- Every object implements **wait()**, **notify()**, and **notifyAll()**

Java Bounded Buffer

```
public synchronized void insert(E item){
    while(count==BUFFER_SIZE)
        this.wait();
    buffer[in]=item;
    in= (in+1) % BUFFER_SIZE;
    count++;
    this.notifyAll();
}

public synchronized E remove(){

    while(count==0)
        this.wait();
    E item = buffer[out];
    out = (out+1) % BUFFER_SIZE;
    count--;
    this.notifyAll();
    return item;
}
```

- Since there is a single (hidden) condition variable, one must call signalALL()
- No need to pass the lock to the calls to wait() since everything is implicit anyway
- You can only call wait() and notify() inside a synchronized method
- Locks are reentrant (you can call a synchronized method from another synchronized method of the same object)
- A thread can hold many locks
- notify() and notifyAll() do nothing if nobody is waiting

Priority Inversion

- Going back toward the OS, we have seen that processes/threads can have different priorities
- Let's just say that a higher priority process, if ready, always runs before a lower priority process (like in priority scheduling)
- **Important:** Processes, even if their code doesn't lead to synchronization problems, use data structures in the kernel that are themselves protected by, e.g., locks
 - Whether you see it or not, your programs do use locks, and cond vars, semaphores, etc., when they run in kernel mode
- Let's say we have three processes: $H > M > L$
 - Resource R (e.g., a linked list in which elements are inserted/removed) is currently in use by process L
 - Process L holds a lock called mutex
 - Process H requires resource R
 - Process H is blocked on lock(mutex)
 - But process M is running, preventing process L from running for a long time
 - So process L can never call unlock(mutex)
- **Priority Inversion:** Process M runs, and runs, while process H is stuck

Priority Inversion Solution

- Most OSes implement a **priority inheritance** mechanism
- A process that accesses a resource needed by a higher priority process inherits that process' priority temporarily
 - Makes the Kernel code a bit more complex
- This solves the example seen in the previous slide
- Read Section 6.5.4 and the “Priority Inversion and the Mars Pathfinder” blurb
 - The program was real-time, so higher-priority processes had better run when they need to!
 - If priority inheritance hadn't been implemented in the kernel of the OS, the pathfinder would have failed

Semaphores

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore **S** – integer variable
- Can only be accessed via two indivisible (atomic) operations

– **wait()** and **signal()**

- Originally called **P()** and **V()**

- Definition of the **wait()** operation

```
wait(S) {  
    while (S <= 0)  
        ; // busy wait  
    S--;  
}
```

- Definition of the **signal()** operation

```
signal(S) {  
    S++;  
}
```

Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
 - Same as a **mutex lock**
- Can solve various synchronization problems
- Consider P_1 and P_2 that require S_1 to happen before S_2

Create a semaphore “**synch**” initialized to 0

P1:

$S_1;$

signal(synch);

P2:

wait(synch);

$S_2;$

- Can implement a counting semaphore **S** as a binary semaphore

Semaphore Implementation

- Must guarantee that no two processes can execute the `wait()` and `signal()` on the same semaphore at the same time
- Thus, the implementation becomes the critical section problem where the `wait` and `signal` code are placed in the critical section
 - Could now have **busy waiting** in critical section implementation
 - But implementation code is short
 - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
 - value (of type integer)
 - pointer to next record in the list
- Two operations:
 - **block** – place the process invoking the operation on the appropriate waiting queue
 - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue
- ```
typedef struct{
 int value;
 struct process *list;
} semaphore;
```

# Implementation with no Busy waiting (Cont.)

```
wait(semaphore *S) {
 S->value--;
 if (S->value < 0) {
 add this process to S->list;
 block();
 }
}
```

```
signal(semaphore *S) {
 S->value++;
 if (S->value <= 0) {
 remove a process P from S->list;
 wakeup(P);
 }
}
```

# Synchronization Concerns

- **Race conditions**
  - Inconsistent program state leading to error or incorrect execution
- **Deadlock**
  - No thread can make progress
- **Starvation**
  - Some threads do not get access to the CPU even though they should
- **Unfairness**
  - Some threads do not get access to the CPU enough compared to other threads
- **Livelock**
  - Constant flip-flop without any progress being made



# Synchronization in Solaris

- Solaris provides:
  - adaptive mutexes
  - condition variables
  - semaphores
  - reader-writer locks
  - turnstiles
- Adaptive mutexes
  - looks at the state of the system and “decides” whether to **spin** or to **block**
  - e.g., if the lock is currently being held by a thread that’s blocked, forget spinning
  - No matter what, long critical sections should be protected by semaphores or cond. variables so that one is certain that there will be no spinning

# Synchronization in Win XP

- The Kernel uses **spin locks** for protection within the Kernel
  - Or interrupt-disabling on single-processor systems
- It ensures that a (kernel) thread holding a spin lock is never preempted
- For user-programs, XP provides **dispatcher objects**
  - mutex locks
  - semaphores
  - event (a.k.a. condition variables)
  - timers(send a signal()) after a lapse of time)

# Synchronization in Linux

- Locking in the Kernel: spin locks and semaphores
  - Spin locks protect only short code sections
  - ON single-core machines, disables kernel preemption
    - Which is allowed only if the current thread does not hold any locks (the kernel counts locks held per thread)
  - (Non-spin) Semaphores used for longer sections of code
- Pthreads
  - (non spin) mutex locks
  - spin locks
  - condition variables
  - read-write locks
  - semaphores

# Conclusion

- Thread Synchronization is an important topic
  - Theory is difficult
  - Practice is difficult
- The future may change this unfortunate situation
  - New “concurrent” languages
  - New ways to think about concurrent programming
  - Help from the compiler
  - Help for the hardware: transaction memories
- Perhaps take CSC-4310 Parallel and Distributed Computing
- Don't forget about the programming assignment!