Concurrent threads/processes (informal)
- Two processes are concurrent if they run at the same time or if their execution is interleaved in any order

Asynchronous
- The processes require occasional synchronization

Independent
- They do not have any reliance on each other

Synchronous
- Frequent synchronization with each other — order of execution is guaranteed

Parallel
- Processes run at the same time on separate processors

Race Conditions
A race condition is a bug:
- The outcome of concurrent threads are unexpectedly dependent on a specific sequence of events.

Example
- Your current bank balance is $1,000.
- Withdraw $500 from an ATM machine while a $5,000 direct deposit is coming in

Withdrawal
- Acquire(transfer_lock)
- Read account balance
- Subtract 500
- Write account balance
- Release(transfer_lock)

Deposit
- Acquire(transfer_lock)
- Read account balance
- Add 5000
- Write account balance
- Release(transfer_lock)

Possible outcomes:
Total balance = $5500 $500 $6000

Synchronization
Synchronization deals with developing techniques to avoid race conditions

Something as simple as
\[ x = x + 1; \]
Compiles to this and may cause a race condition:

<table>
<thead>
<tr>
<th>Address</th>
<th>Label</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>_x</td>
<td>movl %eax, %rdx</td>
</tr>
<tr>
<td>0x04</td>
<td>(%rip)</td>
<td>addl $1, %rdx</td>
</tr>
<tr>
<td>0x08</td>
<td>%eax</td>
<td>movl %eax, _x (%rip)</td>
</tr>
</tbody>
</table>

Potential points of preemption for a race condition

Avoid race conditions with locks
- Grab and release locks around critical sections
- Wait if you cannot get a lock

Critical section:
Region in a program where race conditions can arise

Mutual exclusion:
Allow only one thread to access a critical section at a time

Deadlock:
A thread is perpetually blocked (circular dependency on resources)

Starvation:
A thread is perpetually denied resources

Livelock:
Threads run but with no progress in execution
The Critical Section Problem

Design a protocol to allow threads to enter a critical section

Conditions for a solution

- Mutual exclusion: No threads may be inside the same critical sections simultaneously.
- Progress: If no thread is executing in its critical section but one or more threads want to enter, the selection of a thread cannot be delayed indefinitely.
  - If one thread wants to enter, it should be permitted to enter.
  - If multiple threads want to enter, exactly one should be selected.
- Bounded waiting: No thread should wait forever to enter a critical section.
- No thread running outside its critical section may block others.
- A good solution will make no assumptions on:
  - No assumptions on # processors
  - No assumptions on # threads/processes
  - Relative speed of each thread

Critical sections & the kernel

- Multiprocessors
  - Multiple processes on different processors may access the kernel simultaneously.
  - Interrupts may occur on multiple processors simultaneously.
- Preemptive kernels
  - Preemptive kernel: process can be preempted while running in kernel mode (the scheduler may preempt a process even if it is running in the kernel).
  - Nonpreemptive kernel: processes running in kernel mode cannot be preempted (but interrupts can still occur).
- Single processor, nonpreemptive kernel
  - Free from race conditions.

Solution #1: Disable Interrupts

Disable all system interrupts before entering a critical section and re-enable them when leaving.

Bad!
- Gives the thread too much control over the system.
- Stops time updates and scheduling.
- What if the logic in the critical section goes wrong?
- What if the critical section has a dependency on some other interrupt, thread, or system call?
- What about multiple processors? Disabling interrupts affects just one processor.

Advantage
- Simple, guaranteed to work.
- Was often used in the uniprocessor kernels.

Solution #2: Software Test & Set Locks

Keep a shared lock variable:

```c
while (locked) ;
locked = 1;
/* do critical section */
locked = 0;
```

Disadvantage:
- Buggy! There’s a race condition in setting the lock.

Advantage:
- Simple to understand. It's been used for things such as locking mailbox files.

Solution #3: Lockstep Synchronization

Take turns

```c
// Thread 0
while (turn != 0) ;
critical_section();
turn = 1;

// Thread 1
while (turn != 1) ;
critical_section();
turn = 0;
```

Disadvantage:
- Forces strict alternation; if thread 2 is really slow, thread 1 is slowed down with it. Turns asynchronous threads into synchronous threads.
Software solutions for mutual exclusion

- Peterson’s solution (page 207 of text), Dekker’s, & others

Disadvantages:
- Difficult to implement correctly
- Have to rely on volatile data types to ensure that compilers don’t make the wrong optimizations
- Difficult to implement for an arbitrary number of threads

Let’s turn to hardware for help

Help from the processor

Atomic (indivisible) CPU instructions that help us get locks
- Test-and-set
- Compare-and-swap
- Fetch-and-Increment

These instructions execute in their entirety: they cannot be interrupted or preempted partway through their execution

Test & Set

Set the lock but get told if it already was set (in which case you don’t have it)

```c
int test_and_set(int *x) {
    last_value = *x;
    *x = 1;
    return last_value;
}
```

How you use it to lock a critical section (i.e., enforce mutual exclusion):
```c
while (test_and_set(&locked) == 1) ; /* spin */
/* do critical section */
lock = 0; /* release the lock */
```

Compare & swap (CAS)

Compare the value of a memory location with an old value. If they match then replace with a new value

```c
int compare_and_swap(int *x, int old, int new) {
    int save = *x;
    if (save == old) {
        *x = new;
        return save; /* always return location contents */
    }
}
```

How you use it to grab a critical section:
Avoid the race condition – set locked to 1 only if locked was still set to 0.
```c
while (compare_and_swap(&locked, 0, 1) != 0) ;
/* spin until locked == 0 */
/* if we got here, locked got set to 1 and we have it */
/* do critical section */
locked = 0; /* release the lock */
```

Fetch & Increment

Increment a memory location; return previous value

```c
int fetch_and_increment(int *x) {
    last_value = *x;
    *x = *x + 1;
    return last_value;
}
```

© 2014 Paul Krzyzanowski
**Fetch & Increment**

Check that it’s your turn for the critical section

Ticket lock

```java
ticket = 0; turn = 0;
...
myturn = fetch_and_increment(ticket);
while (turn != myturn);
/* do critical section */
fetch_and_increment(turn);
```

**The problem with spin locks**

- All these solutions require busy waiting
  - Tight loop that spins waiting for a turn: **busy waiting** or **spin lock**
- Nothing useful gets done!
  - Wastes CPU cycles

**Priority Inversion**

- Spin locks may lead to **priority inversion**
- The process with the lock may not be allowed to run!
  - Suppose a lower priority process obtained a lock
    - Higher priority process is always ready to run but loops on trying to get the lock
    - Scheduler always schedules the higher-priority process
  - Priority inversion
    - If the low priority process would get to run & release its lock, it would then accelerate the time for the high priority process to get a chance to get the lock and do useful work
    - Try explaining that to a scheduler!

**Priority Inheritance**

- Technique to avoid priority inversion
- Increase the priority of any process in a critical section to the maximum of any process waiting on any resource for which the process has a lock
- When the lock is released, the priority goes to its normal level

**Spin locks aren’t great**

*Can we block until we can get the critical section?*

```java
public class Lock {
    private int val = UNLOCKED;
    private ThreadQueue waitQueue = new ThreadQueue();

    public void acquire() {
        Thread me = Thread.currentThread();
        while (TestAndSet(val) == LOCKED) {
            waitQueue.waitForAccess(me); // Put self in queue
            me.sleep(); // Put self to sleep
        }
    // Got the lock
    }

    public void release() {
        Thread next = waitQueue.nextThread();
        val = UNLOCKED;
        if (next != null)
            next.ready(); // Wake up a waiting thread
    }
```
Sorry...

- Accessing the wait queue is a critical section
  - Need to add mutual exclusion
- Need extra lock check in acquire
  - Thread may find the lock busy
  - Another thread may release the lock but before the first thread enqueues itself
- This can get ugly!

Semaphores

- Count # of wake-ups saved for future use
- Two atomic operations:
  - `down(sem s) {` 
    - `if (s > 0)` 
    - `s = s - 1;`
    - `else`
    - `sleep on event s`
  - `up(sem s) {` 
    - `if (someone is waiting on s)` 
    - `wake up one of the threads`
    - `else`
    - `s = s + 1;` 

Semaphores

Count the number of threads that may enter a critical section at any given time.
- Each `down` decreases the number of future accesses
- When no more are allowed, processes have to wait
- Each `up` lets a waiting process get in

Producer-Consumer example

- Producer
  - Generates items that go into a buffer
  - Maximum buffer capacity = N
  - If the producer fills the buffer, it must wait (sleep)
- Consumer
  - Consumes things from the buffer
  - If there’s nothing in the buffer, it must wait (sleep)
- This is known as the Bounded-Buffer Problem

Producer-Consumer example

```c
sem mutex=1, empty=N, full=0;
producer() { 
  for (;;) { 
    produce_item[i]; // produce something 
    down[empty]; // decrement empty count 
    enter_item[i]; // put item in buffer 
    up[full]; // +1 full slot 
  } 
}
consumer() { 
  for (;;) { 
    down[full]; // one less item 
    down[mutex]; // start critical section 
    remove_item[i]; // get the item from the buffer 
    up[mutex]; // end critical section 
    up[empty]; // one more empty slot 
    consume_item[i]; // consume it 
  } 
}
```

Readers-Writers example

- Shared data store (e.g., database)
- Multiple processes can read concurrently
- Allow only one process to write at a time
  - And no readers can read while the writer is writing
Readers- Writers example

```c
/* write */
 sơ viết = 1; // critical section used only by the reader
 sơ canh viết = 1; // critical section for N readers vs. 1 writer
 int readcount = 0; // number of concurrent readers
 writer(); {
   for (;;) {
     down(canwrite); // block if we cannot write
     up(canwrite); // end critical section
   }
}

/* read */
/* mut excl */
/* advance */
/* await */
```

Event Counters

Avoid race conditions without using mutual exclusion

An event counter is an integer

Three operations:

- **read(E):** return the current value of event counter E
- **advance(E):** increment E (atomically)
- **await(E, v):** wait until E ≥ v

Producer-Consumer example

```c
#define N 4 // four slots in the buffer
event_counter in=0; // number of items inserted into buffer
event_counter out=0; // number of items removed from buffer

producer(); {
  int item, sequence=0;
  for (;;) {
    produce_item(); // produce something
    sequence++; // item # of item produced
    up(on_event()); // wait until there's room (0≥3), (0≥2), ...
    enter_item(); // put it item in buffer
    advance_item(); // let consumer know there's one more item
  }
}

customer(); {
  int item, sequence=0;
  for (;;) {
    sequence++; // item # we want to consume
    up(on_event()); // wait until that item in present (0≥2)
    consume_item(); // get the item from the buffer
    advance_item(); // let producer know item’s gone
    consume_item(); // consume it
  }
}

Suppose the producer runs for a while, and the consumer does not:

Iteration 1: out=0, sequence=0, await=0, 1-4: continue since 0 ≥ 0⇒0 ≥ 1
Iteration 2: out=0, sequence=1, await=2, 2-4: continue since 0 ≥ 2⇒0 ≥ 1
Iteration 3: out=0, sequence=3, await=3, 3-4: continue since 0 ≥ 3⇒0 ≥ 1
Iteration 4: out=0, sequence=4, await=4, 4-4: continue since 0 ≥ 4⇒0 ≥ 1
Iteration 5: out=0, sequence=5, await=5, 5-4: wait since 0 < 1
```

© 2014 Paul Krzyzanowski
Condition Variables / Monitors

- Higher-level synchronization primitive
- Implemented by the programming language / APIs
- Two operations:
  - `wait(condition_variable)`
    - Block until condition_variable is "signaled"
  - `signal(condition_variable)`
    - Wake up one process that is waiting on the condition variable
    - Also called `notify`

Communicating processes

- Must:
  - Synchronize
  - Exchange data
- Message passing offers:
  - Data communication
  - Synchronization (via waiting for messages)
  - Works with processes on different machines

Message passing

- Two primitives:
  - `send(destination, message)`
  - `receive(source, message)`
- Operations may or may not be blocking

Producer-consumer example

```c
// number of slots in the buffer
#define N 4

consumer() {
  int item, i;
  message m;
  for (i=0; i < N; ++i)
    send(producer, &m);  // send N empty messages
  for (; ; )
    receive(producer, &m);  // get a message with the item
    extract_item(&m, &item);  // take item out of message
    send(producer, &m);  // send an empty reply
    consume_item(&item);  // consume it
}

producer() {
  int item, message m;
  for (; ; )
    produce_item(&m);  // produce something
    send_item_to_consumer(&m);  // send it off
    build_message(&m, item);  // construct the message
  }
```

Messaging: Rendezvous

- Sending process blocked until receive occurs
- Receive blocks until a send occurs

- Advantages:
  - No need for message buffering if on same system
  - Easy & efficient to implement
  - Allows for tight synchronization

- Disadvantage:
  - Forces sender & receiver to run in lockstep
Messaging: Direct Addressing

- Sending process identifies receiving process
- Receiving process can identify sending process
  - Or can receive it as a parameter

Messaging: Indirect Addressing

- Messages sent to an intermediary data structure of FIFO queues
- Each queue is a mailbox
- Simplifies multiple readers

Mailboxes

- Single sender, single reader
- Single sender, multiple readers
- Multiple senders, single reader
- Multiple senders, multiple readers

Other common IPC mechanisms

- Shared files
  - File locking allows concurrent access control
  - Mandatory or advisory
- Signal
  - A simple poke
- Pipe
  - Two-way data stream using file descriptors (but not names)
  - Need a common parent or threads in the same process
- Named pipe (FIFO file)
  - Like a pipe but opened like a file
- Shared memory

Conditions for deadlock

Four conditions must hold

1. Mutual exclusion
   - Only one thread can access a critical section (resource) at a time
2. Hold and wait
   - A thread holds a resource but waits for another resource
3. Non-preemption of resources
   - Resources can only be released voluntarily
4. Circular wait
   - There is a cyclic dependency of threads waiting on resources

Deadlock

- Resource allocation
  - Resource R₁ is allocated to process P₁: assignment edge
  - Resource R₁ is requested by process P₁: request edge
- Deadlock is present when the graph has cycles
Deadlock example

Circular dependency among four processes and four resources leads to deadlock

Dealing with deadlock

• Deadlock prevention
  – Ensure that at least one of the necessary conditions cannot hold

• Deadlock avoidance
  – Provide advance information to the OS on which resources a process will request.
  – OS can then decide if the process should wait
  – *But knowing which resources will be used (and when) is hard! (impossible, really)*

• Deadlock detection
  – Detect when a deadlock occurs and then deal with it

• Ignore the problem
  – Let the user deal with it (most common approach)

The End