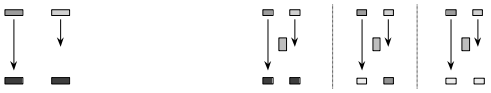
Synchronization

Coordinating the Activity of Mostly Independent Entities

Independent vs. Cooperating Processes

- Independent
  - no shared state
  - deterministic
  - each process can proceed at arbitrary rate
- Cooperating
  - Shared state
    - like a common variable
  - non-deterministic execution
    - hard to reproduce, and subject to race conditions



A Two Process Example

Who Wins?

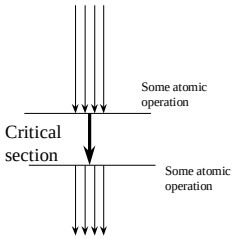
```
int A;
thread_fork(doBlue, 0);
thread_fork(doRed, 0);
```

```
doBlue()
{ A = 1;
  if (A ==1) printf("Black wins");
}
```

```
doRed()
{A = 2;
 if (A == 2) printf("Red wins");
}
```

Programming with MUTUAL EXCLUSION

- Means
  - adding synchronization so you can reason about what your program is doing.
- Synchronized sections are called *Critical Sections*
  - An atomic operation guards access to the critical section.
  - Only one thread at a time can execute in the critical section.
    - others are forced to wait on entry.



Our Two Process Example

```
blueProcess()
{
   if (redProcessNotInCriticalSection()) {
        blueProcessInCriticalSection();
        /* DO CRITICAL SECTION */
         blueProcessNotInCriticalSection();
}
```

This "TEST AND SET" operation must execute atomically. Can not be interrupted.

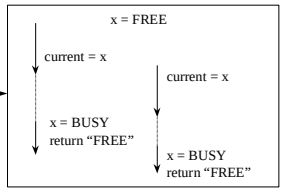
Atomic Operations

- Problem:
  - Must know what is atomic before we can say anything about outcome.
- Defn:
  - An "Atomic" operation is one which executes as though it were not interrupted in either time and space.
    - time is easier to think about
    - but it is really space that we care about

Atomic Test And Set

- Test a value to see if it indicates “available”
- Set the value to indicate available
- Must execute indivisibly

```
TestAndSet x
  int current = x;
  if (current == FREE)
     x = BUSY;
  return current;
```



Where do Atomic Operations Come From?

- The hardware is required to do *something* atomically.
  - More important to know what it is than for it to be something explicit
    - atomic register swap
    - test and set
    - atomic memory load and store
      - x = 32;
      - x = y;
  - With this minimal atomic operation, can build others on top.

More Specifically

- Simple reads and writes of single words are generally atomic.
  - Consider a double word load...
- On a uniprocessor, anything between interrupts is atomic.
- Without SOME atomic operation, you can’t make others
  - can be as simple as an atomic bit set.
- Simple atomic operations can be used to build more complicated ones

Requirements of a Critical Section

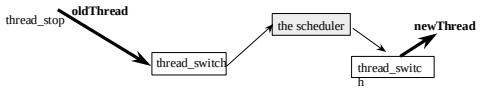
- Mutual exclusion
  - it works
  - if one process is in the CS, then no other is.
- Progress
  - if some process P is not in the critical section, the P can not prevent some other process Q from entering the critical section.
- Bounded Waiting (No starvation)
  - if some process P is waiting on the critical section, then P will eventually enter the critical section.
- Performance
  - The overhead of entering and exiting the critical section is low with respect to the work being done within.

Higher Level Mechanisms for Building Critical Sections

- Semaphores
  - Very primitive. Almost specifying direct start and stop of threads.
  - Simple. Hard to program with, but easy to get the hang of.
- Monitors
  - Higher level than semaphores (language support). More abstract.
- Messages
  - Simple model of communication and synchronization based on atomic transfer of data across a channel.
  - Direct application to distributed systems
  - We’ll do this one last since it’s the easiest (once we see how the others work).

Foundations for Synchronization

- **thread_stop()**
  - takes the currently running thread, saves its state, and switches to another thread.
- **thread_start(thread_t t)**
  - takes the named thread and makes it runnable
- **thread_switch(thread_t t1, thread_t t2)**
  - takes the currently running thread (t1) and saves the current state in t1’s TCB. Restores the previous context from the TCB of t2.



Semaphores

- Abstract data type with an internal counter that is accessed atomically.
  - P(s)
    - wait for s.counter > 0; decrement s.counter.
  - V(s)
    - increment s

```
Semaphore_t s;
int x;
...
P(s);
      X++
V(s);
```

Example: Atomically increment a shared variable x

Binary Semaphores

- Used for mutual exclusion
  - Only one process can access a critical section at a time.
  - To enter the critical section, P on the semaphore.
  - When leaving the critical section, V on the semaphore.
  - Don't forget to V.
- Counter can take on two values: 0,1
  - initially, count is 1.
  - P(s) :
    - if s.count > 0 then s.count = 0
    - if s.count == 0 then wait
  - V(sem): count(sem) = 1;

Counting semaphores

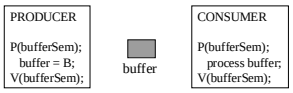
- One "resource" with the mutual exclusion example.
  - the resource is the critical section.
- When you have a lot of resources and you want to hand them out:
  - if there is one, you can have it.
  - otherwise, you must wait.
- Want to use COUNTING SEMAPHORES: [0..N]
  - initially, count == N.
    - P(sem): if count(sem) > 0, count(sem) --;
      - if count(sem) == 0 wait;
    - V(sem): count(sem)++;

Producer/Consumer

- Producer
  - generates new "things" (disk blocks, memory pages, processes, files, etc.)
- Consumer
  - consumes things generated by the producer
- Want to allow producer and consumer to run at different rates
  - no serialization of one behind the other
  - tasks are independent
  - easier to think about

Producer/Consumer

- Introduce an intermediate BUFFER to handle the mismatched rates.
  - Producer deposits items into the buffer
  - Consumer pulls items out of the buffer.
- Need to synchronize access to buffer.



Is this ok?

Synchronizing on the buffer and the rate

- The previous example gave us mutually exclusive access to the buffer, but did not allow us to proceed at mismatched rates.
  - not much better than having one process.
- Introduce more buffers.
  - use counting semaphores, treat each buffer as a resource.
- Semaphores
  - empty: [0,N] --> are there any empty slots?
  - full: [0,N] --> are there any full slots?
  - bufferPool: [0,1] --> can I access the buffer pool now?

Buffer Pool

```
PRODUCER

P(empty);  /* take one
    empty */
P(bufferPool);
  /* put new buffer into pool
    */
V(bufferPool);
V(full);  /* create one full */
```

```
CONSUMER

P(full);
P(bufferPool);
    /* get buffer from pool */
V(bufferPool);
V(empty);
```

Does order of V matter?
Does order of P matter?

Summary

- Concurrent programs involve shared state between independent activities
- Some kind of concurrency control mechanism is required
- Higher level synchronization services can be built on top of lower level ones
  - test and set --> semaphores --> others??