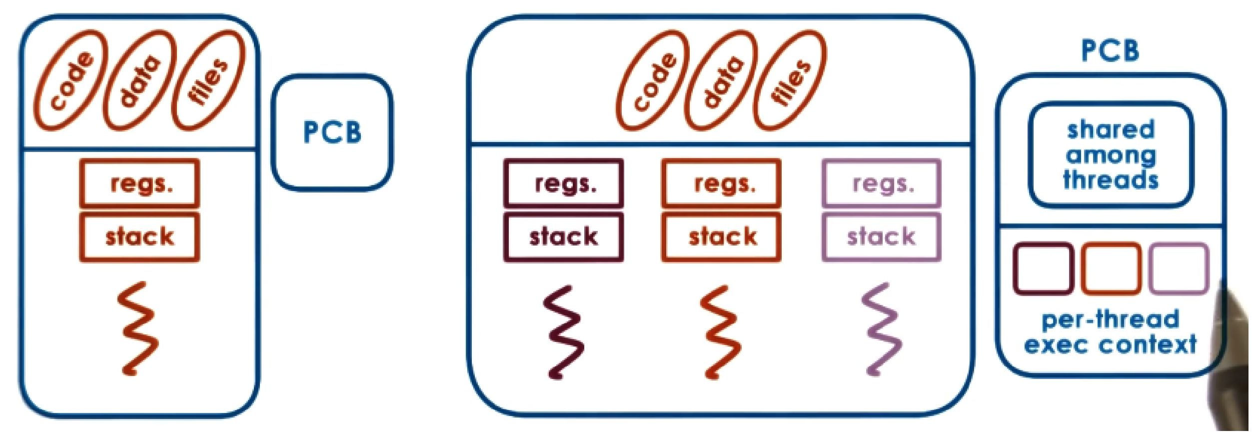
Threads and Concurrency

Process vs Thread

1. A thread is the construct that allows for multiple execution contexts within a single process
    - Active entity - Executing unit of a process
    - Works simultaneously with others - Many threads generally execute at once
    - Requires coordination - Sharing of IO devices, CPUs, memory, etc.
2. Processes are represented by its address space
    - Contains virtual to physical address mappings (code, data, files)
    - Also represented by execution context (stack pointer, program counter)
    - All of this is contained in the Process Control Block
3. Threads are part of the same virtual address space, sharing code, data, files
    - Can operate on different portions of the input, execute different instructions
    - Require separate registers, program counter, stacks for each thread
4. Types of state:
    - Text and data (static state when process first loads)
    - Heap (dynamically created during execution)
    - Stack (grows and shrinks, LIFO queue)



Process vs Thread

Why Are Threads Useful?

1. Parallelization - Can process input much faster
2. Specialization - Can give higher priority to threads that are processing more important input
    - Each thread has its own CPU cache, so the cache remains hotter
3. Efficiency - Lower memory management requirement, cheaper interprocess communication
4. Multithreaded application tends to have smaller memory footprint than multiprocess alternative
    - Threads share address space, so less duplication is required
5. Threads are still useful when # threads > # CPUs
    - if(t_idle > 2 * t_ctx_switch)
    - t_ctx_switch_thread < t_ctx_switch_process
    - Can hide latency associated with IO operations
6. Benefits:
    - Multithreading OS kernel allows OS to support multiple execution contexts
    - Particularly useful when there are multiple CPUs
    - Can run daemons or device drivers

What Do We Need to Support Threads?

1. Data structure to identify threads, keep track of resource usage
2. Mechanisms to create and manage threads
3. Mechanisms to safely coordinate among threads
4. Thread type (thread data structure)
   - Thread ID
   - PC
   - SP
   - Registers
   - Stack
   - Other attributes (priority)
5. Fork (process, arguments)
   - Create a thread (not the same as a UNIX fork system call)
   - t1 = fork(proc, args)
6. Join(thread) - Terminate a thread
   - child.result = join(t1)

Mutual Exclusion

1. Lock that should be used when accessing any data shared among threads
2. Elements:
   - locked?
   - owner
   - blocked__threads
3. Critical section - Portion of code protected by the mutex

Condition Variables

1. Used in conjunction with mutexes to control the behavior of concurrent threads
2. Useful in producer/consumer example
   - Consumer can "wait" on a condition, producers can "signal" the consumer when true

```
// consumer: print_and_clear
Lock(m) {
   while (my_list.not_full())
Wait(m, list_full);
   my_list.print_and_remove_all();
} // unlock;
```

```
// producers: safe_insert
Lock(m) {
   my_list.insert(my_thread_id);
   if my_list.full()
     Signal(list_full);
} // unlock;
```

Producer/Consumer Example

Condition Variables API

1. Condition type
2. wait(mutex, condition)
   - Mutex is automatically released and reacquired on wait

3. signal(condition)
    - Notify only one thread waiting on condition
4. broadcast(condition)
    - Notify all waiting threads

```
Mutex counter_mutex; Condition read_phase, write_phase; int resource_counter = 0;
```

```
// READERS
Lock(counter_mutex) {
  while(resource_counter == -1)
    Wait(counter_mutex, read_phase);
  resource_counter++;
} // unlock;
// ... read data ...
Lock(counter_mutex) {
  resource_counter--;
  if(readers == 0)
    Signal(write_phase);
} // unlock;
```

```
// WRITER
Lock(counter_mutex) {
  while(resouce_counter != 0)
    Wait(counter_mutex, write_phase);
  resource_counter = -1;
} // unlock;
// ... write data ...
Lock(counter_mutex) {
  resource_counter = 0;
  Broadcast(read_phase);
  Signal(write_phase);
} // unlock;
```

Reader/Writer Example

Critical Section Structure

1. lock(mutex)
    - while(!predicate_indicating_access_ok)
        - wait(mutex, cond_var)
    - update state => update predicate
    - signal and/or broadcast (cond_var_with_correct_waiting_threads)
2. unlock(mutex)

Common Pitfalls

1. Keep track of mutex/condition variables are associated with which resource
    - mutex_tpye m1; // mutex for file1
2. Check that you are always (and correctly) using lock and unlock
    - Some compilers can generate warnings and errors
3. Use a single mutex to access a single resource
4. Check that you are signaling the correct condition
5. Check that you are not using signal when broadcast is needed
    - Only one thread will proceed on signal, others will wait (possibly indefinitely)
6. Thread execution order is not guaranteed by the order they are signalled

Spurious Wake Ups

1. Definition: Threads are signalled while the mutex they require is still held elsewhere
2. Still correct, but hurts performance
3. Occurs when a broadcast/signal call is made while the mutex is still held

Deadlocks

1. Definition: Two or more competing threads waiting on each other to complete, but neither do
2. Can occur when two threads lock the same mutexes in different orders
3. Maintaining a lock order can prevent this
    - Acquiring mutex B implies mutex A is already acquired
4. A cycle in the wait graph is necessary and sufficient for a deadlock to occur
    - Edges from thread waiting on a response to thread owning a resource
5. What can we do about it?
    - Deadlock prevention - Lock order, but can be expensive
    - Deadlock detection and recovery - Rollback, generally less expensive
    - Apply the ostrich algorithm - Do nothing, reboot if deadlock happens

Kernel-Level vs User-Level Threads

1. User-level threads must be associated with a kernel-level thread
    - Scheduling is handled by the scheduler in the kernel
2. One-to-one Model - One kernel thread per user thread
    - Pros:
        - OS sees/understands threads, synchronization, blocking
    - Cons
        - Must go to OS for all operations (expensive)
        - OS may have limits on policies and number of threads
        - Portability
3. Many-to-one Model - Multiple user-level threads mapped to a single kernel-level thread
    - Pros:
        - Totally portable, doesn't depend on OS limits and policies
    - Cons:
        - OS has no insight into application needs
        - OS may block entire process if one user-level thread blocks on IO
4. Many-to-many Model - Some user-level threads mapped one-to-one, others one-to-many
    - Pros:
        - Can be the best of both worlds
        - Can have bound (one-to-one) and unbound (many-to-one) mappings
    - Cons:
        - Requires coordination between user- and kernel-level thread managers
5. Scope of Multithreading
    - System scope - System-wide thread management by OS-level thread managers (CPU scheduler)
    - Process scope - User-level library manages threads within a single process

Multithreading Patterns

1. Boss-Workers
2. Pipeline
3. Layered

Boss/Workers Pattern

1. Boss assigns work to workers
2. Workers perform entire task assigned to them
3. Throughput of system is limited by boss thread -> must keep boss efficient
4. Throughput = 1 / boss_time_per_order
5. Boss can assign work by…
    - Directly signalling specific worker
        - Pros - Workers don't need to synchronize

- Cons - Boss must track what each worker is doing, so throughput is lower
- Placing work in producer/consumer queue
  - Pros - Boss doesn't need to know details about workers
  - Cons - Queue synchronization

6. Producer/consumer queue gives better throughput
7. How many workers is enough?
   - On demand - Add as needed
   - Pool of workers - Number of threads is generally dynamically allocated
8. Pros - Simplicity
9. Cons - Thread pool management, locality
10. Instead of creating all workers equal, we can specialize workers for certain tasks
    - Pros - Better locality and quality of service management
    - Cons - Load balancing becomes more difficult
11. Throughput = time_to_complete * ceiling(num_jobs / num_workers)

Pipeline Pattern

1. Threads assigned to one subtask in the system
2. Entire task is executed as a pipeline of threads
3. Multiple tasks are executed concurrently in the system at different pipeline stages
4. Throughput - Only as fast as the slowest stage of the pipeline
   - Can assign more threads from a pool to the slowest stage
5. Shared buffer used for communication between stages
6. Pros - Specialization and locality
7. Cons - Balancing and synchronization overheads
8. Throughput = time_to_complete_one + ((num_jobs - 1) * time_to_complete_last_stage)

Layered Pattern

1. Each layer is assigned a group of related tasks
2. End-to-end task must pass up and down through all layers
3. Pros - Specialization and locality, but less fine-grained than pipeline
4. Cons - Not suitable for all applications, synchronization between layers