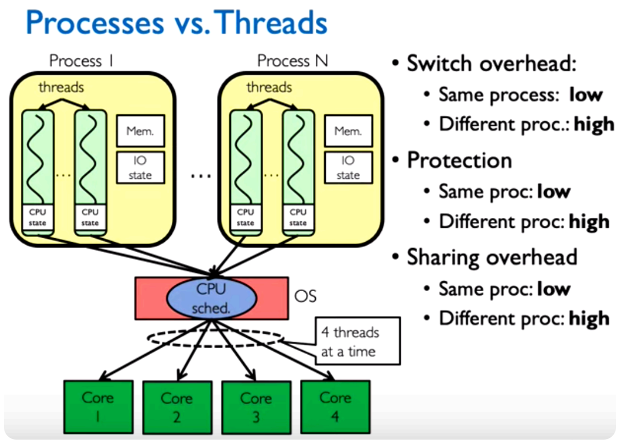
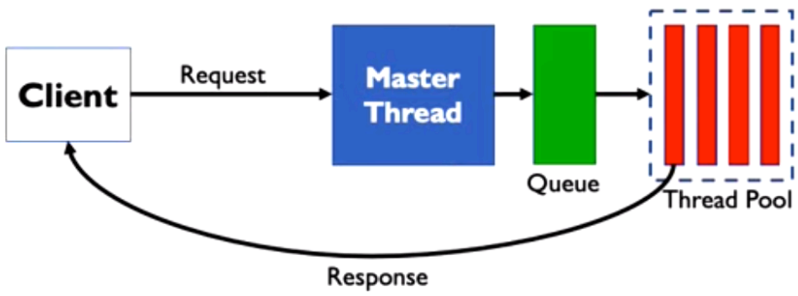
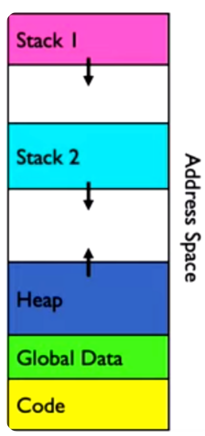
- Thread
    - single execution sequence (basic unit) working inside a protection boundary (i.e. process's address space), a separately schedulable task
    - kernel inherently uses threads
    - has register state and stack living in address space of a process
        - local state = (its stack), shared state (static data and heap)
        - thread state (registers: sp, ip) kept in TCB when thread is not running
    - protection
        - can have 1 or many threads per protection domain (i.e. process)
        - single threaded user program: 1 thread per process (PINTOS)
        - multi threaded ...: multiple threads sharing same data structures, isolated from other user programs
        - multi threaded kernel: multiple threads sharing kernel data structures, capable of privileged instructions
    - motivation
        - OS needs to handle multiple things at once [MTAO] (processes, interrupts, background system maintenance)
        - servers (multiple connections), parallel programs (better performance), UI's (to achieve responsiveness), network and disk bound programs (to hide latency) also need to handle MTAO

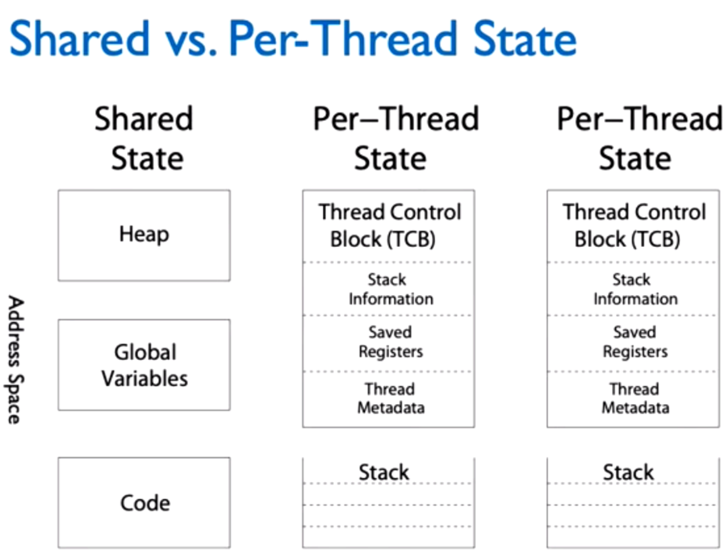


-
    - processor is really fast => for slow system work (e.g. disk access), we can keep the processor busy doing other tasks
    - thread voluntarily giving up control
        - I/O
            - e.g. keyboard listens for keypress; while it waits, let CPU do other important work
        - waiting for a signal from another thread
            - thread makes syscall to wait
        - thread executes thread_yield
            - manually relinquishes CPU; but calling thread gets put on ready queue immediately



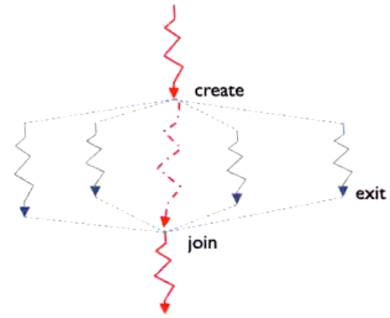
-
    - switching threads (nanoseconds) is MUCH CHEAPER than switching processes (microseconds)
        - no need to change address space (page table)
        - start a new process => isolation/protection
            - just start a new thread => performance
    - e.g. multi-threaded server
        - loop: accept new connection, fork a thread/process to service it
        - if too many requests => might run out of memory (thread stacks), schedulers can't handle too many threads
            - can use **thread pools**: fixed/bounded number of worker threads, allocated in advance (=> no thread creation overhead)
            - have a queue of pending task requests => wait for a thread to execute on





-
    - vocabulary
        - multiprocessing: multiple cores
        - multiprogramming: multiple jobs per process
        - multithreading: multiple threads per process
- Thread vs Process State
    - process wide state
        - address space, memory contents (global variables, heap)

        - I/O bookkeeping
    - thread-local state
        - CPU registers including program counter
        - execution stack
        - TCB
    - shared state across all threads
        - each thread has their own stack
        - kernel manages TCB for each thread
    - thread stacks must be big enough, but small enough to fit in user memory space
        - how much space should we leave between stacks (so they don't overwrite each other)




```
int pthread_create(pthread_t *thread,
      const pthread_attr_t *attr,
      void *(*start_routine)(void*), void *arg);
```

- thread is created executing *start_routine* with *arg* as its sole argument. (return is implicit call to pthread_exit)

```
void pthread_exit(void *value_ptr);
```

- terminates and makes *value_ptr* available to any successful join

```
int pthread_join(pthread_t thread, void **value_ptr);
```

- suspends execution of the calling thread until the target *thread* terminates.
- On return with a non-NULL *value_ptr* the value passed to pthread_exit() by the terminating thread is made available in the location referenced by *value_ptr*.

- Preempting a thread
    - if a thread never voluntarily gives up control => dispatcher/kernel gains control via **interrupts**
        - signals from HW or SW to stop whatever thread is running and jump to kernel
    - set timer every ms to switch threads
    - context switches between processes <= same idea => between threads
        - except don't have to change address space between intra-process threads
- Start with ThreadRoot
    - who is passed a function that grows/initializes the thread stack
- User-level multithreading: **pthreads** (corresponds to fork for processes)
    - when thread exits, it can pass some result to a ptr that is made available to any successful join (e.g. by a calling thread)
    - pthread_join puts calling thread to sleep until target thread calls pthread_exit (and terminates)
        - but the target thread's stack may not have been deallocated just yet
    - **fork thread pattern**:
        - main thread forks collection of sub threads, passing them args to work on
        - => joins with them, collecting the results
- Correctness with concurrent threads
    - non-determinism:
        - scheduler can run threads in any order and switch threads at any time
    - for independent threads: there is no shared state, so this is ok
    - with shared state between multiple threads, we can run into data inconsistencies
        - race condition: thread A races against thread B (outcome of data depends on order of execution)
    - atomic operations
        - operation that runs to completion or not at all
        - need some atomic modifications (R/W) to allow threads to work together
    - mutual exclusion - ensure only one thread does a particular thing at a time on the data (1 thread excludes others)
    - critical section - code that exactly one thread can execute at once (result of mutual exclusion), atomic code
    - use locks to provide mutual exclusion in critical sections
        - lock - an object only one thread can hold at a time
            - a synchronization variable that provides mutual exclusion

Critical section

```
int common = 162;
pthread_mutex_t common_lock = PTHREAD_MUTEX_INITIALIZER;

void *threadfun(void *threadid)
{
  long tid = (long)threadid;
  pthread_mutex_lock(&common_lock);
  int my_common = common++;
  pthread_mutex_unlock(&common_lock);

  printf("Thread #%lx stack: %lx common: %lx (%d)\n", tid,
         (unsigned long) &tid,
         (unsigned long) &common, my_common);
  pthread_exit(NULL);
}
```

        - lock associated with some shared state; thread needs to hold lock in order to access that state
        - makes the shared object "thread safe"
        - operations on the shared object are a critical section
      - has 2 atomic operations: acquire (wait until lock free => grab), release (unlock, wake up waiters)
    - e.g. threadfun is a function executed by multiple threads
      - use pthread_mutex_t to create a mutually exclusive object
      - essentially used to create an atomic critical section of code
    - semaphores (i.e. railway gate) are a kind of generalized lock
      - can be any non-negative integer, can be initialized to anything >= 0
      - has 2 atomic operations:
        - P() or down(): waits for semaphore to go positive => decrements it by 1
        - V() or up(): increments semaphore by 1 => wakes up any waiting P
      - e.g. implemented as a **lock**
        - if semaphore is initialized to 1 => down locks it => up releases the lock (see right image)
      - e.g. can be used to **thread_join**
        - the semaphore can't go negative => down (in ThreadJoin) must wait until semaphore is incremented to 1 by the up call in ThreadFinish
        - this type of semaphore is called a condition variable
      - intuition for semaphores: what do you need to wait for? and what variable can you set to 0 when you need to wait?

Mutual Exclusion: (Like lock)

- Called a "binary semaphore"

```
initial value of semaphore = 1;
semaphore.down();
 // Critical section goes here
semaphore.up();
```

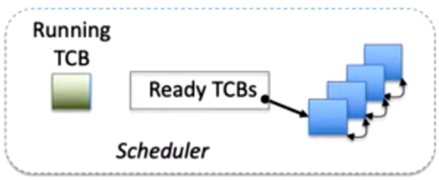
Signaling other threads, e.g. **ThreadJoin**

```
Initial value of semaphore = 0

                                    ThreadFinish {
                                        semaphore.up();
ThreadJoin {                        }
   semaphore.down();
}
```

Think of *down* as wait() operation

- Implementing locks
    - single core
      - disable interrupts while holding lock to ensure atomic operation (guarantee no interference in the middle of critical section)
      - naive:
        - x86 has instructions cli and sti to enable/disable interrupts
        - acquire: by disabling interrupts, release: by re-enabling interrupts
        - terrible idea, because if we acquire and then the thread has an infinite loop => no way for system to exit because no interrupts allowed!
          - => can't do any I/O either!




```
Acquire(*lock) {
   disable interrupts;
   if (lock->value == BUSY) {
      put thread on lock's wait_Q
      "i.e, Go to sleep"
      allow a ready thread to run
   } else {
      lock->value = BUSY;
   }
   enable interrupts;
}
```

```
Release(*lock) {
   disable interrupts;
   if (any TCB on lock wait_Q) {
      "i.e., lock busy";
      take thread off wait queue
      Place on ready queue;
   } else {
      lock->value = FREE;
   }
   enable interrupts;
}
```

      - we only want to disable interrupts over a tiny window (to ensure atomic access to the lock itself)
        - critical section is only the block in between the disable and enable of interrupts (very short)
        - value indicates the lock's status
        - this lock signals whether a thread has permission to access a data structure
      - acquire and release are themselves basic atomic operations
        - disable interrupts => accessing the lock state doesn't itself run into synchronization issues
      - then for a thread's critical section (atomically executed code), acquire --> do atomic stuff --> release
        - for atomic actions, surround with lock acquire and release
      - a lock is a value (FREE or BUSY) + list of threads (waiting on that value)
      - if lock busy => an acquiring thread is put on the lock's waiting queue
        - it suspends itself => allows switch to another runnable thread (by enabling kernel interrupts)
        - when some thread releases this lock => the acquiring thread is put back on the scheduler's ready queue (removed from the lock's wait queue)

        - note that we re-enable interrupts after a context switch (re-enabled by the next thread to run)
        - if lock is busy => at least 1 thread on wait queue
            - first thread on wait Q is the current thread with the lock?
- synchronization variables - data structure used to coordinate concurrent access to shared state
    - e.g. locks and condition variables
    - both can be implemented with semaphores
        - built with atomic read-modify-write instructions
- **shared objects** use synchronization variables to coordinate multiple threads' access to shared state
    - shared objects should be allocated on the heap (not in a function's local stack which can disappear after the returns)

- Threads hold illusion of infinite number of processors (each thread can get its own processor)
- Current PCB
    - pid, name, etc
    - TCBs (thread objects)
        - place to save registers when not running
        - thread status
        - links to form lists
    - Thread stack
    - Lock object (per thread)
        - for any lock used by its kernel thread
    - current working directory
    - file descriptors/handles for open files