1. Processes and Threads

- **Function call**
  - single entry/exit point
  - alloc/dealloc on stack
  - caller/callee register save
- **Cooperative/user-level thread switch**
  - multiple entry/exit points
  - save/restore all registers
  - switch stacks (save/restore SP)
- **Thread switch**
  - more expensive, but close
  - user-level/co-op switch, spawn: very efficient

1b. Mesa Activation records

- **Multithreading**
  - don't need multiple stack segments -> base & bound!
  - easy to spawn new thread
  - needs synchronized malloc (probably have this anyway)
  - more complex alloc/dealloc; fragmentation, collisions...
- **Function calls**
  - stack is *much* simpler/faster; no sync, no fragmentation
  - don't need to switch stacks... don't really use stacks (except for temps, possibly)

2. Synchronization/deadlock

- You *can* implement semaphores with locks.
  - busy-waiting solution (basically ch. 6.5)
    ```c
    void down(semaphore *s) {
      success = 0;
      while (!success) {
        lock(s->lock);
        if (value == 0) {
          unlock(s->lock);
          yield();
        }
        else success = 1;
      }
      s->value--;
      unlock(s->lock);
      return;
    }
    ```
- If one thread can release another's lock...
  ```c
typedef struct { lock wait, protect; int value; } sem...

void down(semaphore *s) {
  lock(s->wait);
  /* s->wait can only be acquired if value > 0 */
  lock(s->protect);
  s->value--;
  if (s->value > 0) unlock(s->wait);
  unlock(s->protect);
}

void up(semaphore *s) {
  lock(s->protect);
  s->value++;
  if (s->value == 1) unlock(s->wait);
  unlock(s->protect);
}
```

2a. Semaphores w/locks (pt. 3)

- Implement low-level primitives using locks
  ```c
  lock swap_lock;
  void atomic_swap(int *a, int *b) {
    int c;
    lock(swap_lock);
    c = *b; *b = *a; *a = c;
    unlock(swap_lock);
  }
  ```
  Then use them to implement semaphores...

2b. Broken/deadlocking code

- basically deadlock example from lecture, but no unlock!
  ```c
  void thread_move(queue_t *q1, queue_t *q2) {
    lock(q1->lock); lock(q2->lock);
    push_first(q1, pop_last(q2));
  }
  ```
  ```c
  fix 1: serialize the whole thing (lock for thread_move)
  fix 2 (improved?): acquire all locks simultaneously
  fix 2a: order lock acquisitions (e.g. by comparing pointers)
  fix 3 (better design?): put the locks in push_first() and pop_last()
  - (means we can't call queue routines at interrupt time)
2c. Transactional Memory

- Printf()
  - No locks, can call at interrupt level!
  - Might not work depending on rollback
  - If rollback restores memory, cannot “undo” a write (or read!) to/from I/O space

Real world TM generally cache-like
  - Doesn’t work on I/O accesses

Printf is a long routine
  - Long transactions → likely to abort

3. Scheduling

- Parallel workloads
  Most people got that multiple (e.g. disk + GPU + CPU-intensive) workloads could run in parallel

- Serial workloads
  Stuff that must run sequentially (e.g. compile then link):

- No benefit workloads
  Workloads that you can overlap but not run in parallel (worsens average completion time.)

- Gang scheduling
  Share 64 processors between a 2p job and a 64p job?
  62 idle processors 50% of time!

4. Linking

- Code:
  ```c
  int a = 9;
  static int b = 1;
  extern int c;
  main(int argc, char **argv)
  {
    a = 2;
    add(&a, &c);
    printf("the result is %d\n", a);
  }
  ```

- Relocation table:

5. Virtual memory

- 0x1000 a = (int *) 0x5100; TLB miss (0x1000)
  0x1004 b = *a; TLB miss (0x5100), page fault (0x5100)/zero fill
  0x1008 a = (int *) 0x3300;
  0x100C c = *a; TLB miss (0x3300), page fault (0x3300)/swap in
  0x1010 a = b + c;
  0x1014 f = (void (*)()) 0x2200;
  0x1018 sp = (int *)0xFFF0;
  0x101C *sp = 0x1028;
  TLB miss (0xFFF0)
  0x1020 sp -= 1;
  0x1024 goto f; TLB miss (0x2200)
  (Instruction at 0x2200 is missing, but presumably unimportant)
  0x1028 a = (int *) 0x6400
  0x2208 *a = b; TLB miss (0x6400)
  0x220C f = *(sp); TLB miss (0xFFE0); f = <undefined>!
  0x2210 goto f; (Program could die here since f could point to anything, but... we apparently got lucky - undefined value was 0x1028) TLB miss (0x1028)
  0x1028 a = (int *) 0x2800;
  0x102C b = *a; TLB miss (0x2800)
  0x1030 a = (int *) 0x4800;
  0x1034 *a = c; SegV (0x4600)

6. Course comments/suggestions

- Thanks for the feedback!!
- Good:
  People (say they) like/enjoy OS/Pintos/CS140! 😊
- Improvements:
  - Improve slides (diagrams (Adobe), Comic Sans MS, clarity)
  - SCPD: later office hours (we have this)
  - Project size/workload/structure (summer, future?)
  - Demos, explaining code, examples (inc. apps)
  - Speak slowly
  - More/better project info/details
- More suggestions? Send them to us!! (or to one of us)