Intro to Threads

- **Two approaches to concurrency**
  - Asynchronous I/O (lab)
  - Threads (today's lecture)
- **Threaded multifinger:**
  - Run many copies of `finger` simultaneously

```
for (int i = 1; i < argc; i++)
  thread_create (finger, argv[i]);
```

How to share CPU amongst threads

- **Each thread has execution state:**
  - Stack, program counter, registers, condition codes, etc.
- **Switch the CPU amongst the threads**
  - Save away execution state of one, load up that of next
- **When to switch?**
  - Current thread can no longer use the CPU (waiting for I/O)
  - Current thread has had CPU for too long (preemption)
  - Scheduler maintains lists of runnable/running/waiting threads

Thread package API

- `tid create (void (*fn) (void *), void *arg);`
  - Create a new thread, run `fn` with `arg`
- `void exit ();`
  - Destroy current thread
- `void join (tid thread);`
  - Wait for thread `thread` to exit

Synchronization primitives

- `void lock (mutex_t m);`
  `void unlock (mutex_t m);`
  - Only one thread acuires `m` at a time, others wait
  - **All global data must be protected by a mutex!**
- `void wait (mutex_t m, cond_t c);`
  - Atomically unlock `m` and sleep until `c` signaled
- `void signal (cond_t c);`
  `void broadcast (cond_t c);`
  - Wake one/all users waiting on `c`

Example: Taking job from work queue

```
job *job_queue;
mutex_t job_mutex;
cond_t job_cond;
void workthread (void *) {
  job *j;
  for (;;) {
    lock (job_mutex);
    while (!(j = job_queue))
      wait (job_mutex, job_cond);
    job_queue = j->next;
    unlock (job_mutex);
    do (j);
  }
}
```

Example: Adding job to work queue

```
void addjob (job *j) {
  lock (job_mutex);
  j->next = job_queue;
  job_queue = j;
  signal (job_cond);
  unlock (job_mutex);
}
```

- **Atomic release and wait was necessary in** `workthread`

Other thread package features

- Alerts – cause exception in a thread
- Trylock – don't block if can't acquire mutex
- Timedwait – timeout on condition variable
- Shared locks – concurrent read accesses to data
- Thread priorities – control scheduling policy
- Thread-specific global data

Implementing shared locks

```
struct sharedlk {
  int i; mutex_t m; cond_t c;
};
void AcquireExclusive (sharedlk *sl) {
  lock (sl->m);
  while (sl->i) { wait (sl->m, sl->c); }
  sl->i = -1;
  unlock (sl->m);
}
void AcquireShared (sharedlk *sl) {
  lock (sl->m);
  while (sl->i < 0) { wait (sl->m, sl->c); }
  sl->i++;
  unlock (sl->m);
}
```

shared locks (continued)

```
void ReleaseShared (sharedlk *lk) {
  lock (sl->m);
  if (!--i) signal (lk->c);
  unlock (sl->m);
}
void ReleaseExclusive (sharedlk *lk) {
  lock (sl->m);
  i = 0;
  broadcast (lk->c);
  unlock (sl->m);
}
```

- **Must deal with starvation**

Deadlock

- **Mutex ordering:**
  - A locks m1, B locks m2, A locks m2, B locks m1
  - How to avoid?
- **Similar deadlock with condition variables**
  - Suppose resource 1 managed by c_1, resource 2 by c_2
  - A has 1, waits on $c2$, B has 2, waits on $c1$
- **Mutex/condition variable deadlock:**

```
- lock (a); lock (b); while (!ready) wait (b, c);
  unlock (b); unlock (a);
- lock (a); lock (b); ready = true; signal (c);
  unlock (b); unlock (a);
```

Bad to hold locks when crossing abstraction barriers!

Detecting deadlock

- **Static approaches (hard)**
- **Threads package can keep track of locks held**
- **Program grinds to a halt**
  - Examine with debugger, find lock order problem
- **Threads package can deduce partial order**
  - For each lock acquired, order with other locks held
  - If cycle occurs, abort with error
  - Detects potential deadlocks even if they do not occur

Data races

- **Example: modify global `++x` without mutex**
  - Might compile to: load, add 1, store
  - Bad interleaving changes result: load, load, . . .
- **Even single instructions can have races**
  - E.g., i386 allows single instruction `addl $1,_x`
  - Not atomic on MP without `lock` prefix!
- **Even reads dangerous on some architectures**
- **But sometimes cheating buys efficiency**

```
if (!initialized) {
  lock (m);
  if (!initialized) { initialize (); initialized = 1; }
  unlock (m);
}
```

Detecting data races

- **Static methods (hard)**
- **Debugging painful—race might occur rarely**
- **Instrumentation—modify program to trap memory accesses**
- **Lockset algorithm (eraser) particularly effective:**
  - For each global memory location, keep a "lockset"
  - On each access, remove any locks not currently held
  - If lockset becomes empty, abort: No mutex protects data
  - Catches potential races even if they don't occur

Implementing user-level threads

- **Allocate a new stack for reach tread `create`**
- **Keep a queue of runnable threads**
- **Replace networking system calls (read/write/etc.)**
  - If operation would block, switch and run different thread
- **Schedule periodic timer signal (`setitimer`)**
  - Switch to another thread on timer signals (preemption)

Example: upt (in ~class/src/)

- **Per-thread state in thread control block structure**

```
typedef struct tcb {
  unsigned long md_esp;          /* Stack pointer of thread */
  char *t_stack;                 /* Bottom of thread's stack */
  /* ... */
};
```

- **Machine-dependent thread-switch function:**
  - `void thread_md_switch (tcb *current, tcb *next);`
- **Machine-dependent thread initialization function:**

  -
    ```
    void thread_md_init (tcp *t,
         void (*fn) (void *), void *arg);
    ```

i386 thread_md_switch

```
pushl %ebp; movl %esp,%ebp          # Save frame pointer
pushl %ebx; pushl %esi; pushl %edi  # Save callee-saved regs

movl 8(%ebp),%edx      # %edx = thread_current
movl 12(%ebp),%eax     # %eax = thread_next
movl %esp,(%edx)       # %edx->md_esp = %esp
movl (%eax),%esp       # %esp = %eax->md_esp

popl %edi; popl %esi; popl %ebx     # Restore callee saved regs
popl %ebp                           # Restore frame pointer
ret                                 # Resume execution
```

i386 thread_md_init

```
void thread_md_init (tcb *t, void (*fn) (void *), void *arg) {
  u_long *sp = (u_long *) (t->t_stack + thread_stack_size);

  /* Set up a callframe to thread_begin */
  *--sp = (u_long) arg;     *--sp = (u_long) fn;
  *--sp = (u_long) t;       *--sp = 0;  /* No return address */

  /* Now set up saved registers for switch.S */
  *--sp = (u_long) thread_begin;  /* return address */
  *--sp = 0;  /* ebp */     *--sp = 0;   /* ebx */
  *--sp = 0;  /* esi */     *--sp = 0;   /* edi */

  t->t_md.md_esp = (mdreg_t) sp;
}
```

- **Swich will call** `thread_begin (fn, arg);`

Implementing synchronization

- **Atomic instructions (e.g. `xchg %eax,(%edx)`)**
  - Typically expensive, even on uniprocessor
- **Can "cheat" on uniprocessor**
  - Set "do not interrupt" bit
  - If timer interrupt arrives, set "interrupted" bit
  - Manipulate mutex data structure
  - Clear DNI bit
  - If interrupted bit set, yield

Limitations of user-level threads

- **Some system calls still block—stop all threads**
  - E.g., disk I/O not asynchronous
- **Page faults block all threads**
- **Technique *can* be extended to multiprocessor**
  - Hard to run as many threads as CPUs

Implementing kernel level threads

- **Start with process abstraction in kernel**
- **Strip out unnecessary features**
  - Same address space
  - Same file table
- **Faster than a process, but still very heavy weight**