System calls for using TCP

Client	Server
	`socket` – make socket
	`bind` – assign address
	`listen` – listen for clients
`socket` – make socket	
`bind` – assign address (optional)	
`connect` – connect to listening socket	
	`accept` – accept connection
`write` – send data	`read` – receive data
`read` – receive data	`write` – send data

- **Anything red might block, waiting for network**
  - Obviously bad for applications that need concurrency

Non-blocking I/O

- **Use `fcntl` to set `O_NONBLOCK` flag on descriptor**
- **Non-blocking semantics of system calls:**
  - `read` immediately returns -1 with `errno` `EAGAIN` if no data
  - `write` may not write all data, or may return `EAGAIN`
  - `connect` may "fail" with `EINPROGRESS` (or may succeed, or may fail with real error like `ECONNREFUSED`)
  - `accept` may fail with `EAGAIN` if no pending connections

How to know when to read/write?

```
struct pollfd {
  int fd;          /* file descriptor */
  short events;    /* Events you are interested in */
  short revents;   /* Events that have happened (results) */
};

int poll(struct pollfd *fds, nfds_t nfds, int timeout);

/* Some possible events: */
#define POLLIN     0x0001  /* Can read fd without blocking */
#define POLLOUT    0x0004  /* Can write fd without blocking */
#define POLLERR    0x0008  /* Error on fd (only in revents) */
#define POLLHUP    0x0010  /* ``Hangup'' has occurred on fd */
```

- **Note: BSD used select to achieve same thing**
  - Most OSes support both `select` and `poll` today

`epoll`

- **Newer Linux provides** `epoll`
- **Interface allows more efficient implementation**
  - Register interest with `epoll_ctl` syscall
  - Wait with `epoll_wait` syscall
  - Kernel doesn't have to re-scan `pollfd` array on each wait
- **New option bits reduce calls to** `epoll_ctl`
  - `EPOLLONESHOT` – only wait for event once
  - `EPOLLET` – "edge triggered" (as opposed to level triggered)
- `epoll` **is Linux specific**
  - But BSD has kqueue/kevent which is similar idea

`epoll` interface

```
typedef union epoll_data {
    int fd;
    /* ... */
} epoll_data_t;

struct epoll_event {
    __uint32_t events;      /* Epoll events */
    epoll_data_t data;      /* User data variable */
};

int epoll_create(int size);
int epoll_ctl(int epfd, int op, int fd,
              struct epoll_event *event);
int epoll_wait(int epfd, struct epoll_event *events,
               int maxevents, int timeout);
```

Asynchronous programming model

- **Many non-blocking file descriptors in one process**
  - Wait for pending I/O events on file many descriptors
  - Each event triggers some *callback* function
- **E.g., build "callback harness":**

```
/* Register callback for when fd is readable or writable */
void cb_add (int fd, int write, void (*fn)(void *), void *arg);

/* Unregister callback */
void cb_free (int fd, int write);

/* Loop forever checking callbacks */
void cb_check (void);
```

Simplified example

```
struct state {
  int fd;
  /* ... */
};

void doit (void) {
  struct state *st = malloc (sizeof (*st));
  st->fd = create_new_tcp_socket ();
  connect (st->fd, &someplace, sizeof (someplace));
  cb_add (st->fd, 1, doit_2, st);
}
static void doit_2 (void *_st) {
  struct state *st = _st;
  write (st->fd, "request\n", 8);
  cb_free (st->fd, 1);
  cb_add (st->fd, 0, doit_3, st);
}
static void doit_3 (void *_st) {
  struct state *st = _st;
  /* read more from st->fd until you get full response */
}
```

Syntactic sugar

- **Problem: Need state from one callback to next**
- **E.g., C++ can implement `wrap` that bundles a function with its arguments**

```
callback<void, int>::ref errwrite = wrap (write, 2);
(*errwrite) ("hello", 5);  // calls write (2, "hello", 5);
```

- **Possible to build large event-driven apps this way**
  - E.g., I have built large library to do this
  - Debugging features include recording where callbacks created to facilitate tracing
- **Google reportedly does similar things**

Intro to Threads

- **Threads: most popular abstraction for concurrency**
  - Lighter-weight abstraction than processes
  - All threads in one process have same memory, file desc., etc.
  - Allows one process to use multiple CPUs
- **Example: threaded web server:**
  - Service many clients simultaneously

```
for (;;) {
  fd = accept_client ();
  thread_create (service_client, &fd);
}
```

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/wait necessary in `workthread`, otherwise:**
  - `workthread` checks queue, releases lock
  - `addjob` adds job to queue, signals `job_mutex`
  - `workthread` waits for signal that was already delivered

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

Moral: Bad to hold locks when crossing abstraction barriers!

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., `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);
}
```

Implementing user-level threads

- **Allocate a new stack for reach thread `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

- **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 (tcb *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 kernel level threads

- **Start with process abstraction in kernel**
- **Strip out unnecessary features**
  - Same address space
  - Same file table
  - (Plan9's `rfork` actually allows individual control)
- **Faster than a process, but still very heavy weight**