More Client/Server Programming

Thread Programming

- fork() is expensive (time, memory)
- Interprocess communication is hard.
- Threads are 'lightweight' processes:
  - one process can contain several threads of execution.
  - all threads execute the same program (different stages).
  - all threads share instructions, global memory, open files, and signal handlers.
  - each thread has own thread ID, stack, program counter and stack pointer, errno, signal mask.
  - threads can communicate with shared memory.
  - threads have special synchronization mechanisms.

Thread Programming

- POSIX threads (pthreads): standard for Unix
- OS must support it (Linux)
- Programs must be linked with -lpthread

Pthreads

- Creating a thread:

  #include <pthread.h>

  *int pthread_create(pthread_t *tid, pthread_attr_t *attr, void *(*start_routine)(void *), void *arg);*

  - tid: thread id
  - attr: options
  - start_routine: function to be executed
  - arg: parameter to thread

Pthreads

- Stopping a pthread: a thread stops when
  - the process stops,
  - the parent thread stops,
  - its start_routine function return,
  - or it calls pthread_exit:

  #include <pthread.h>

  *void pthread_exit(void *retval);*

Pthreads

- Threads must be waited for:

  #include <pthread.h>

  *int pthread_join(pthread_t tid, void **status);*

Pthreads Example

```
#include <pthread.h>

void *func(void *param) {
  int *p = (int *) param;
  printf("This is a new thread (%d)\n", *p);
  return NULL;
}
int main () {
  pthread_t id;
  int x = 100;

  pthread_create(&id, NULL, func, (void *) &x);
  pthread_join(id, NULL);
}
```

Pthreads

- A thread can be joinable or detached.
- Detached: on termination all thread resources are released, does not stop when parent thread stops, does not need to be pthread_join()ed.
- Default: joinable (attached), on termination thread ID and exit status are saved by OS.

Pthreads

- Creating a detached thread:

```
pthread_t id;
pthread_attr_t attr;

pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr,
    PTHREAD_CREATE_DETACHED);
pthread_create(&id, &attr, func, NULL);
```

- pthread_detach()

Pthreads

- A thread can join another:
  *int pthread_join (pthread_t tid, void ** status);*
- Call waits until specified thread exits.

Pthreads

```
int counter = 0;
void *thread_code (void *arg) {
  counter++;
  printf("Thread %u is number %d\n",
    pthread_self(), counter);
}
main () {
  int i; pthread_t tid;
  for (i = 0; i < 10; i++)
    pthread_create(&tid, NULL, thread_code, NULL);
}
```

Pthread

- Mutual exclusion:
  pthread_mutex_t counter_mtx =
  PTHREAD_MUTEX_INITIALIZER;
- Locking (blocking call):
  *pthread_mutex_lock(pthread_mutex_t *mutex);*
- Unlocking:
  *pthread_mutex_unlock(pthread_mutex_t *mutex);*

Thread Pool

- A server creates a thread for each client. No more than n threads can be active (or n clients can be serviced). How can we let the main thread know that a thread terminated and that it can service a new client?

Possible Solutions

- pthread_join?
  - kinda like *wait()*.
  - requires thread id, so we can wait for thread xy, but not for the 'next' thread
- Global variables?
  - thread startup:
    - acquire lock on the variable
    - increment variable
    - release lock
  - thread termination:
    - acquire lock on the variable
    - decrement variable
    - release lock

Main Loop?

```
active_threads = 0;
// start up first n threads for first n clients
// make sure they are running
while (1) {
  // have to lock/release active_threads;
  if (active_threads < n)
    // start up thread for next client
  busy_waiting(is_bad);
}
```

Condition Variables

- Allow one thread to wait/sleep for event generated by another thread.
- Allows us to avoid busy waiting.

```
pthread_cond_t foo =
  PTHREAD_COND_INITIALIZER;
```

- Condition variable is ALWAYS used with a mutex.

```
pthread_cond_wait(pthread_cond_t *cptr,
  pthread_mutex_t *mptr);

pthread_cond_signal(pthread_cond_t *cptr);
```

Condition Variables

- Each thread decrements active_threads when terminating and calls *pthread_cond_signal()* to wake up main loop.
- The main thread increments active_threads when a thread is started and waits for changes by calling *pthread_cond_wait*.
- All changes to active_threads must be 'within' a mutex.
- If two threads exit 'simultaneously', the second one must wait until the first one is recognized by the main loop.
- Condition signals are NOT lost.

Condition Variables

```
int active_threads = 0;
pthread_mutex_t at_mutex;
pthread_cond_t at_cond;

void *handler_fct(void *arg) {
  // handle client
  pthread_mutex_lock(&at_mutex);
  active_threads--;
  pthread_cond_signal(&at_cond);
  pthread_mutex_unlock(&at_mutex);
  return();
}
```

Condition Variables

```
active_threads = 0;
while (1) {
  pthread_mutex_lock(&at_mutex);
  while (active_threads < n) {
    active_threads++;
    pthread_start(...);
  }
  pthread_cond_wait(&at_cond, &at_mutex);
  pthread_mutex_unlock(&at_mutex);
}
```

Condition Variables

- Multiple 'waiting' threads: signal wakes up exactly one, but not specified which one.
- pthread_cond_wait atomically unlocks mutex.
- When handling signal, pthread_cond_wait atomically re-acquires mutex.
- Avoids race conditions: a signal cannot be sent between the time a thread unlocks a mutex and begins to wait for a signal.

Error Handling

- In general, systems calls return a negative number to indicate an error:
  - we often want to find out what error
  - servers generally add this information to a log
  - clients generally provide some information to the user

extern int errno;

- Whenever an error occurs, system calls set the value of the global variable errno.
  - you can check errno for specific errors
  - you can use support functions to print out or log an ASCII text error message

errno

- errno is valid only after a system call has returned an error.
  - system calls don't *clear* errno on success
  - if you make another system call you may lose the previous value of **`errno`**
    - **`printf`** makes a call to **`write`**!

Error Codes

#include <errno.h>

- Error codes are defined in errno.h

EAGAIN	**EBADF**	**EACCESS**
EBUSY	**EINTR**	**EINVAL**

…

Support Routines

In stdio.h:

*void perror(const char *string);*

In string.h:

*char *strerror(int errnum);*

Using Wrappers

```
int Socket( int f,int t,int p) {
  int n;
  if ( (n=socket(f,t,p)) < 0 )) {
    perror("Fatal Error");
    exit(1);
 }
  return(n);
}
```

Fatal Errors

- How do you know what should be a fatal error (program exits)?
  - common sense.
  - if the program can continue – it should.
  - example – if a server can't create a socket, or can't bind to it's port - there is no sense in continuing…

Server Models

- Iterative servers: process one request at a time.
- Concurrent server: process multiple requests simultaneously.
- Concurrent: better use of resources (service others while waiting) and incoming requests can start being processed immediately after reception.
- Basic server types:
  - Iterative connectionless.
  - Iterative connection-oriented.
  - Concurrent connectionless.
  - Concurrent connection-oriented.

Iterative Server

```
int fd, newfd;
while (1) {
  newfd = accept(fd, ...);
  handle_request(newfd);
  close(newfd);
}
```

- simple
- potentially low resource utilization
- potentially long waiting queue (response times high, rejected requests)

Concurrent Connection-Oriented

1. Master: create a socket, bind it to a well-known address.
2. Master: Place the socket in passive mode.
3. Master: Repeatedly call accept to receive next request from a client, create a new slave process/thread to handle the response.
4. Slave: Begin with a connection passed from the master.
5. Interact with client using this connection (read request, send response).
6. Close the connection and exit.

One Thread Per Client

```
void sig_chld(int) {
  while (waitpid(0, NULL, WNOHANG) > 0) {}
  signal(SIGCHLD, sig_chld);
}

int main() {
  int fd, newfd, pid;
  signal(SIGCHLD, sig_chld);
  while (1) {
    newfd = accept(fd, ...);
    if (newfd < 0) continue;
    pid = fork();
    if (pid == 0) { handle_request(newfd); exit(0); }
    else {close(newfd); }
  }
}
```

Process Pool

```
#define NB_PROC 10
void recv_requests(int fd) {
  int f;
  while (1) {
    f = accept(fd, ...);
    handle_request(f);
    close(f);
  }
}

int main() {
  int fd;
  for (int i=0; i<NB_PROC; i++) {
    if (fork() == 0) recv_requests(fd);
  }
  while (1) pause();
}
```

select() Approach

- Single process manages multiple connections.
- Request treatment needs to be split into non-blocking stages.
- Data structure required to maintain state of each concurrent request.

select() Approach

1. Create a socket, bind to well-known port, add socket to list of those with possible I/O.
2. Use select() to wait for I/O on socket(s).
3. If 'listening' socket is ready, use accept to obtain a new connection and add new socket to list of those with possible I/O.
4. If some other socket is ready, receive request, form a response, send back.
5. Continue with step 2.

select()

*int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);*

- nfds: highest number assigned to a descriptor.
- block until >=1 file descriptors have something to be read, written, or timeout.
- set bit mask for descriptors to watch using FD_SET.
- returns with bits for ready descriptor set: check with FD_ISSET.
- cannot specify amount of data ready.

fd_set

- void FD_ZERO(fd_set *fdset);
- void FD_SET(int fd, fd_set *fdset);
- void FD_CLR(int fd, fd_set *fdset);
- int FD_ISSET(int fd, fd_set *fdset);

- Create fd_set.
- Clear it with FD_ZERO.
- Add descriptors to watch with FD_SET.
- Call select.
- When select returns: use FD_ISSET to see if I/O is possible on each descriptor.

Example (simplified)

```
int main(int argc, char *argv[]) {
  /* variables */
  s = socket(...) /* create socket */
  sin.sin_family = AF_INET;
  sin.sin_port = htons(atoi(argv[1]));
  sin.sin_addr.s_addr = INADDR_ANY;
  bind (s, ...);
  listen(s,5);
  tv.tv_sec = 10;
  tv.tv_usec = 0;
  FD_ZERO(&rfds);
  if (s > 0) FD_SET(s, &rfds);
```

Example (contd)

```
while (1) {
   n = select(FD_SETSIZE, &rfds, NULL, NULL, &tv);
   if (n == 0) printf("Timeout!\n");
   else if (n > 0) {
     if (FD_ISSET(s, &rfds)) {
       t = 0;
       while (t = accept(...) > 0) {
         FD_SET(t, &rfds);
       }
     }
```

Example (contd)

```
for (i = ...) {
      if (FD_ISSET(i, &rfds)) {
        handle_request(i);
      }
    }
  ...
```

- *handle_request:* reads request, sends response, closes socket if client done, calls FD_CLR