CSE 451 11au
Module 2 - Multithreaded Programming

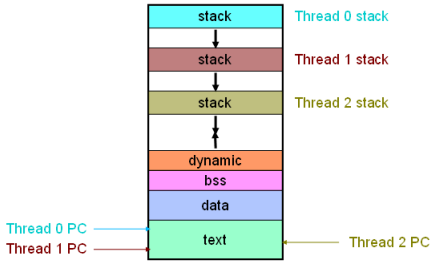
Part 1

Brief introduction to threads

What Is a Thread?

- A thread is an execution context
  - All the state required to execute according to the rules you're accustomed to
- "Process" vs. "Thread"
  - Saying "process" indicates you want to highlight protection boundaries
    - E.g., a virtual address space
  - Saying "thread" indicates you want to highlight concurrency / cooperation
    - The threads of a computation share an address space

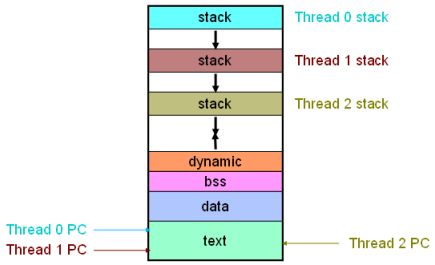
The Virtual Address Space



Execution Context

- A program counter
  - Where is the thread currently executing?
- A stack
  - Procedure invocation history and
  - Local variables
- Additionally, some metadata is specific to each thread
  - e.g., the tid (thread id)

The Virtual Address Space



Implications

- A process can use no more cores at a time than it has threads
- (Procedure) local variables are thread local
  - As always, local variables created by entry into a procedure are accessible only to that execution of that procedure

Uses of Threads

- Parallelism
  - Goal: Higher performance via execution on multiple cores
- Concurrency
  - Goal: Make it easier to write correct code
    - Threads offer a kind of modularity, much as procedures and packages do
    - The modularity in this case is "execution state"

Rest of this module

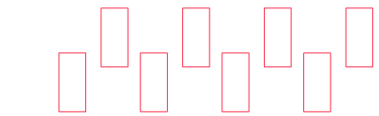
- We'll be most interested in concurrency
  - It would be impossible to write a functional OS without it
  - It would be much harder to write many user-level applications as well
- We'll be looking at a particular thread package
  - pthreads (POSIX threads)

Part 2

Concurrency

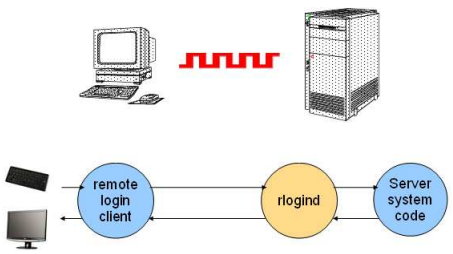
Basic pthreads

Why Threads?



Many things are easier to do with threads
Many things run faster with threads

A Simple Example: rlogind



Life Without Threads

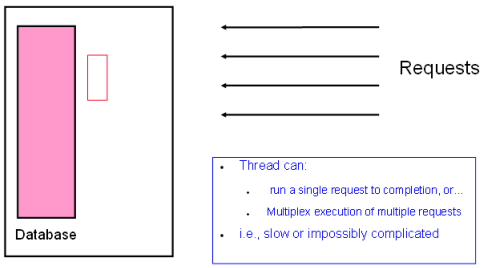
```
login4(int r_in, int r_out, int l_in, int l_out) {
        fd_set in = 0, out;
        int want_l_write = 0, want_r_write = 0;
        int want_l_read = 1, want_r_read = 1;
        int eof = 0, size, fsize, wret;
        char fbuf[BSIZE], tbuf[BSIZE];

        fcntl(r_in, F_SETFL, O_NONBLOCK);
        fcntl(r_out, F_SETFL, O_NONBLOCK);
        fcntl(l_in, F_SETFL, O_NONBLOCK);
        fcntl(l_out, F_SETFL, O_NONBLOCK);

        while(!eof) {
                FD_ZERO(&in);
                FD_ZERO(&out);
                if(want_l_read)
                        FD_SET(l_in, &in);
                if(want_r_read)
                        FD_SET(r_in, &in);
                if(want_l_write)
                        FD_SET(l_out, &out);
                if(want_r_write)
                        FD_SET(r_out, &out);

                select(MAXFD, &in, &out, 0, 0);

                if FD_ISSET(l_in, &in)) {
                        if((size = read(l_in, tbuf, BSIZE)) > 0) {
                                want_l_read = 0;
                                want_r_write = 1;
                        } else
                                eof = 1;
                }
                if (FD_ISSET(r_in, &in)) {
                        if((size = read(r_in, tbuf, BSIZE)) > 0) {
                                want_r_read = 0;
                                want_l_write = 1;
                        } else
                                eof = 1;
                }
                if (FD_ISSET(l_out, &out)) {
                        if((wret = write(l_out, fbuf, fsize)) == fsize) {
                                want_r_read = 1;
                                want_l_write = 0;
                        } else if (wret >= 0)
                                fsize -= wret;
                        else
                                eof = 1;
                }
                if (FD_ISSET(r_out, &out)) {
                        if((wret = write(r_out, tbuf, tsize)) == tsize) {
                                want_l_read = 1;
                                want_r_write = 0;
                        } else if (wret >= 0)
                                tsize -= wret;
                        else
                                eof = 1;
                }
        }
        }
```


Life With Threads

```
incoming(int r_in, int l_out) {
        int eof = 0;
        char buf[BSIZE];
        int size;

        while (!eof) {
                size = read(r_in, buf, BSIZE);
                if (size <= 0)
                        eof = 1;
                if (write(l_out, buf, size) <= 0)
                        eof = 1;
        }
}
```

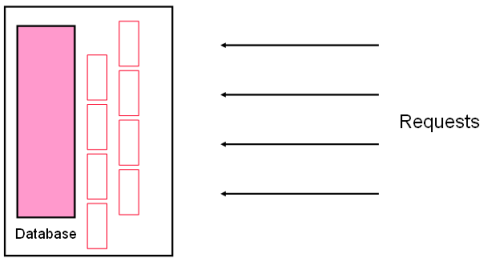
```
outgoing(int l_in, int r_out) {
        int eof = 0;
        char buf[BSIZE];
        int size;

        while (!eof) {
                size = read(l_in, buf, BSIZE);
                if (size <= 0)
                        eof = 1;
                if (write(r_out, buf, size) <= 0)
                        eof = 1;
        }
}
```


Single-Threaded Database Server



- Thread can:
  - run a single request to completion, or...
  - Multiplex execution of multiple requests
- i.e., slow or impossibly complicated

Multithreaded Database Server



Creating a Thread

```
start_servers( ) {
        pthread_t thread;
        int i;
        for (i=0; i<nr_of_server_threads; i++)
                pthread_create(&thread,         // thread ID
                        0,      // attributes (0 means defaults)
                        server, // start routine
                        argument);      // arg to start routine
}

void *server(void *arg) {
        while(1) {
    /* get and handle request */
        }
}
```


Complications

```
rlogind(int r_in, int r_out, int l_in, int l_out) {
    pthread_t in_thread, out_thread;
    pthread_create(&in_thread,
            0,
            incoming,
            r_in, l_out);       // Can't do this ...
    pthread_create(&out_thread,
            0,
            outgoing,
            l_in, r_out);       // Can't do this …

    /* How do we wait till they're done? */
}
```


#1: Multiple Arguments

```
typedef struct {
     int first, second;
} two_ints_t;

rlogind(int r_in, int r_out, int l_in, int l_out) {
     pthread_t in_thread, out_thread;
     two_ints_t in={r_in, l_out}, out={l_in, r_out};
     pthread_create(&in_thread,
      0,
      incoming,
      &in);
     ...
}
```


#2: When Are Threads Done?

```
rlogind(int r_in, int r_out, int l_in, int l_out) {
     pthread_t in_thread, out_thread;
     two_ints_t in={r_in, l_out}, out={l_in, r_out};

     pthread_create(&in_thread, 0, incoming, &in);
     pthread_create(&out_thread, 0, outgoing, &out);

     pthread_join(in_thread, NULL);
     pthread_join(out_thread, NULL);
}
```


Thread Termination

- Threads terminates by:
  - `pthread_exit((void *) value);`
    or
  - `return((void *) value);`
    from its entry point procedure
- Creating thread gets return value:
  - `pthread_join(thread, (void **) &value);`

Sometimes You Don't Want To Wait: Detached Threads

```
start_servers( ) {
     pthread_t thread;
     int i;
     for (i=0; i<nr_of_server_threads; i++) {
   pthread_create(&thread, 0, server, 0);
   pthread_detach(thread);
     }
     ...
}

server( ) {
     ...
}
```


Thread Attributes

```
pthread_t thread;
pthread_attr_t thr_attr;

pthread_attr_init(&thr_attr);

...
/* establish some attributes */
...

pthread_create(&thread, &thr_attr, startroutine, arg);

...
```


Stack Size

```
pthread_t thread;
pthread_attr_t thr_attr;

pthread_attr_init(&thr_attr);
pthread_attr_setstacksize(&thr_attr, 20*1024*1024);

...

pthread_create(&thread, &thr_attr, startroutine,
arg);

...
```


pthread Program: C = AB

```
#include <stdio.h>
#include <pthread.h>
#include <string.h>

#define M       3
#define N       4
#define P       5

int A[M][N];
int B[N][P];
int C[M][P];

void *matmult(void *);

main( ) {
int i;
pthread_t thr[M];
int error;

/* initialize the matrices ... */

...
```


matrix multiply (cont.)

```
// create the worker threads – one per row
for (i=0; i<M; i++) {
      if (error = pthread_create( &thr[i], 0, matmult, (void *)i)) {
            fprintf(stderr, "pthread_create: %s", strerror(error));
            exit(1);
  }
}

// wait for workers to finish their jobs
for (i=0; i<M; i++)
  pthread_join(thr[i], 0)
      /* print the results ... */
}
```


matrix multiply (cont.)

```
// each thread executes this routine
void *matmult(void *arg) {
      int row = (int)arg;  // gens a warning on 64-bit
systems
      int col;
      int i;
      int t;

      for (col=0; col < P; col++) {
            t = 0;
    for (i=0; i<N; i++) {
      t += A[row][i] * B[i][col];
        }
        C[row][col] = t;
      }
      return(0);
}
```


Compiling It

```
Linux% gcc -o mat mat.c -pthread
```

Failure to specify the pthreads library results in a linker error.

