Concurrent Programming

Prof. Jinkyu Jeong(jinkyu@skku.edu)
TA – Jinhong Kim(jinhong.kim@csl.skku.edu)
TA – Seokha Shin(seokha.shin@csl.skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu

Echo Server Revisited

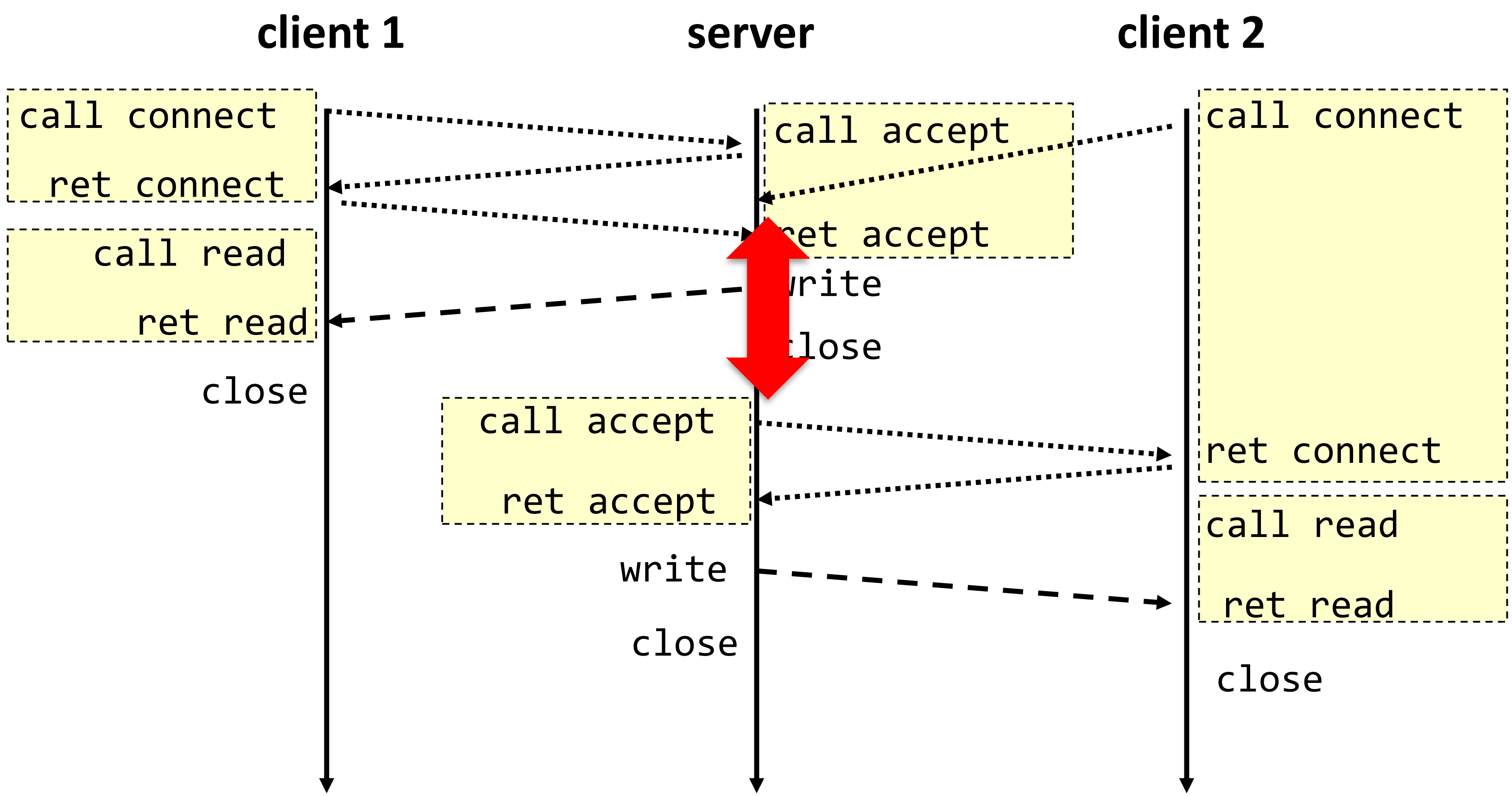
```
int main (int argc, char *argv[]) {
   ...
   listenfd = socket(AF_INET, SOCK_STREAM, 0);

   bzero((char *)&saddr, sizeof(saddr));
   saddr.sin_family = AF_INET;
   saddr.sin_addr.s_addr = htonl(INADDR_ANY);
   saddr.sin_port = htons(port);
   bind(listenfd, (struct sockaddr *)&saddr, sizeof(saddr));

   listen(listenfd, 5);
   while (1) {
      connfd = accept(listenfd, (struct sockaddr *)&caddr, &clen);
      while ((n = read(connfd, buf, MAXLINE)) > 0) {
         printf ("got %d bytes from client.\n", n);
         write(connfd, buf, n);
      }
      close(connfd);
   }
}
```

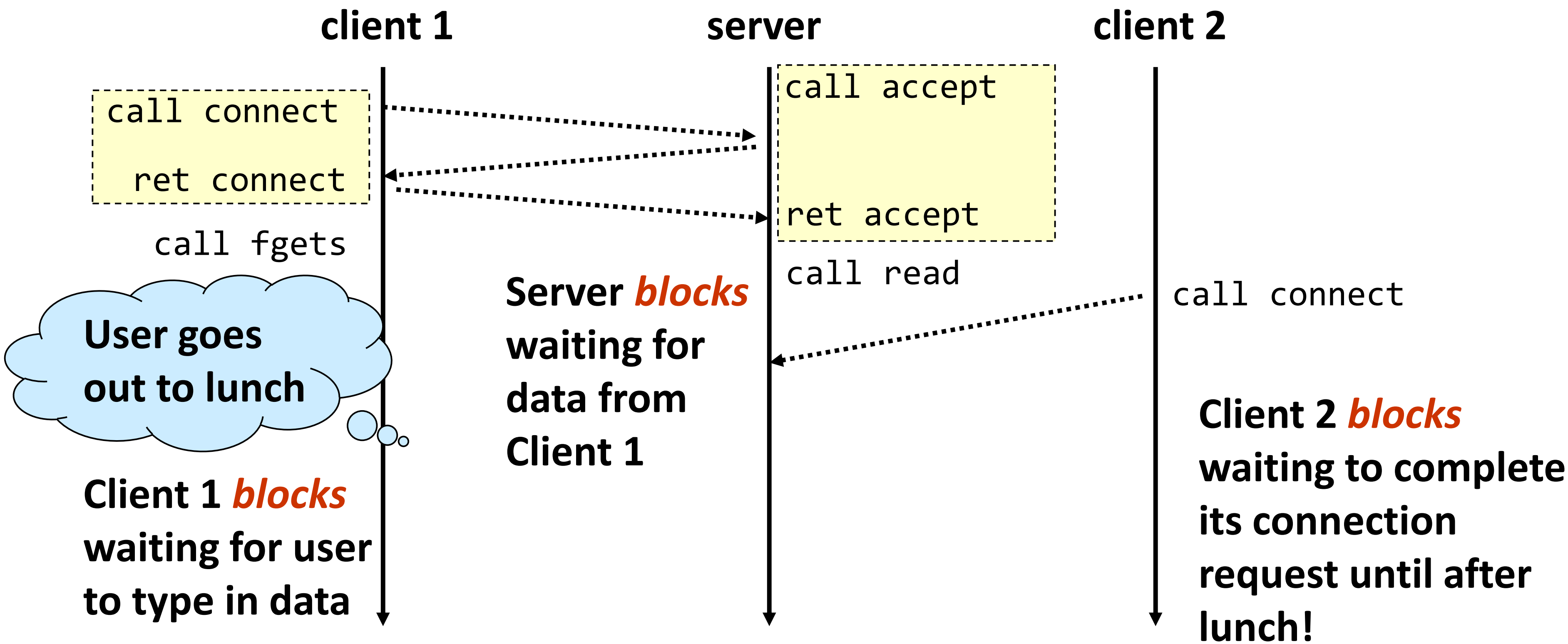
Iterative Servers (1)

- One request at a time



Iterative Servers (2)

- **Fundamental flaw**



- **Solution: use concurrent servers instead**
    - Use multiple concurrent flows to serve multiple clients at the same time.

Creating Concurrent Flows

- **Processes**
  - Kernel automatically interleaves multiple logical flows.
  - Each flow has its own private address space.
- **Threads**
  - Kernel automatically interleaves multiple logical flows.
  - Each flow shares the same address space.
  - Hybrid of processes and I/O multiplexing
- **I/O multiplexing with select()**
  - User manually interleaves multiple logical flows
  - Each flow shares the same address space
  - Popular for high-performance server designs.

Concurrent Programming

Thread-based

SKKU UNIVERSITY

Traditional View

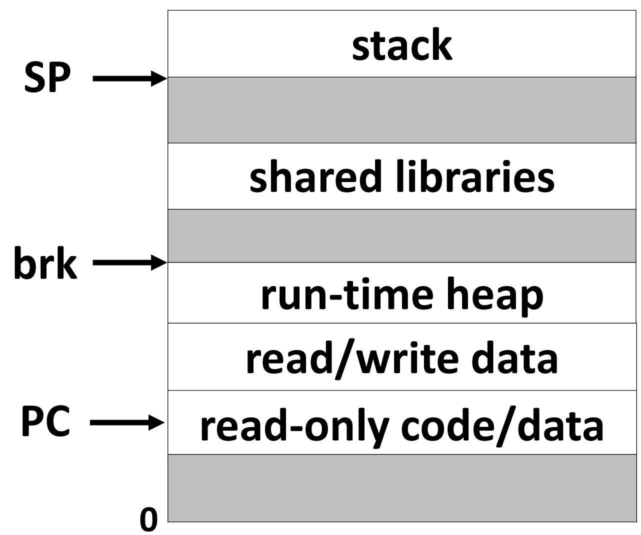
- Process = process context + address space

Process context

Program context:
- Data registers
- Condition codes
- Stack pointer (SP)
- Program counter (PC)

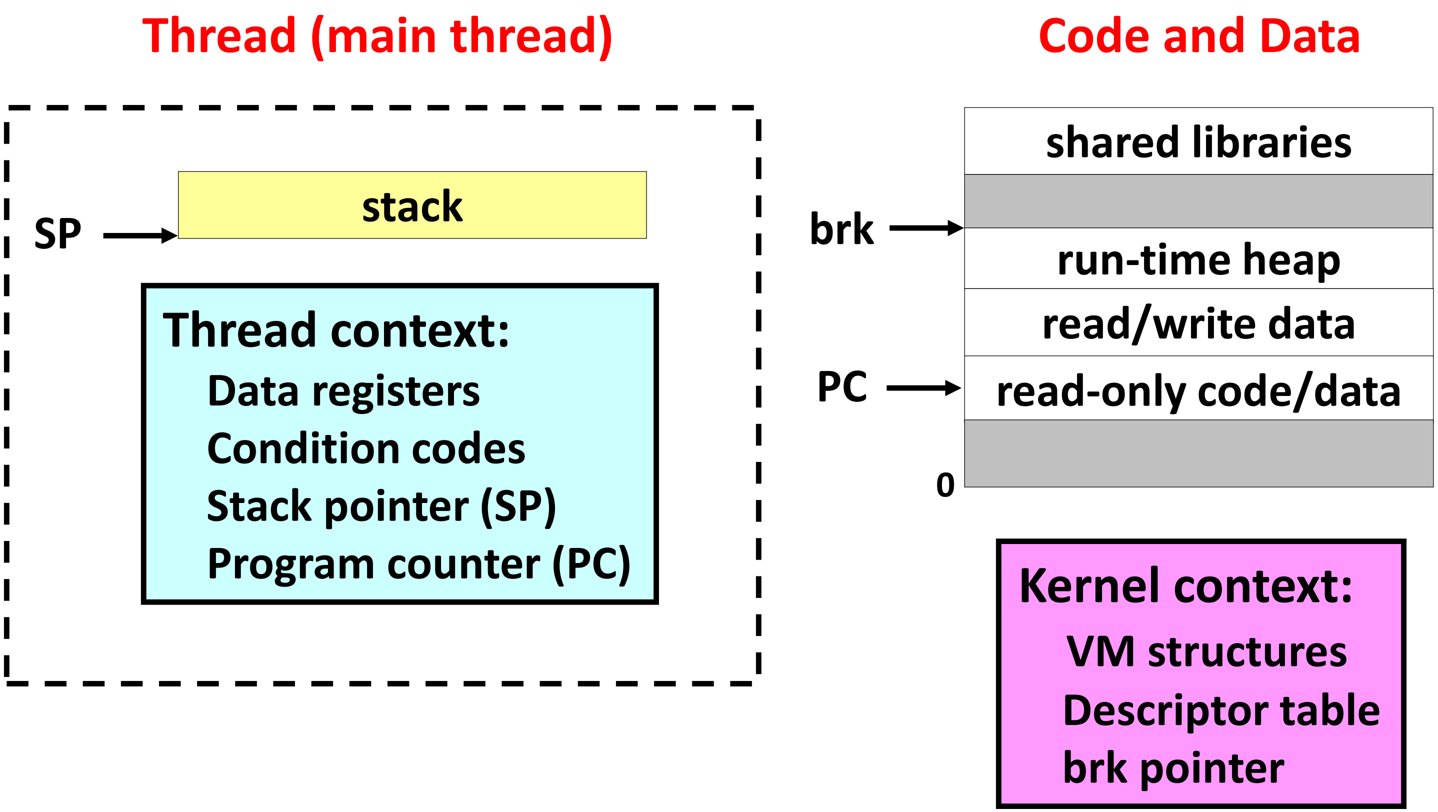
Kernel context:
- VM structures
- Descriptor table
- brk pointer

Code, data, and stack



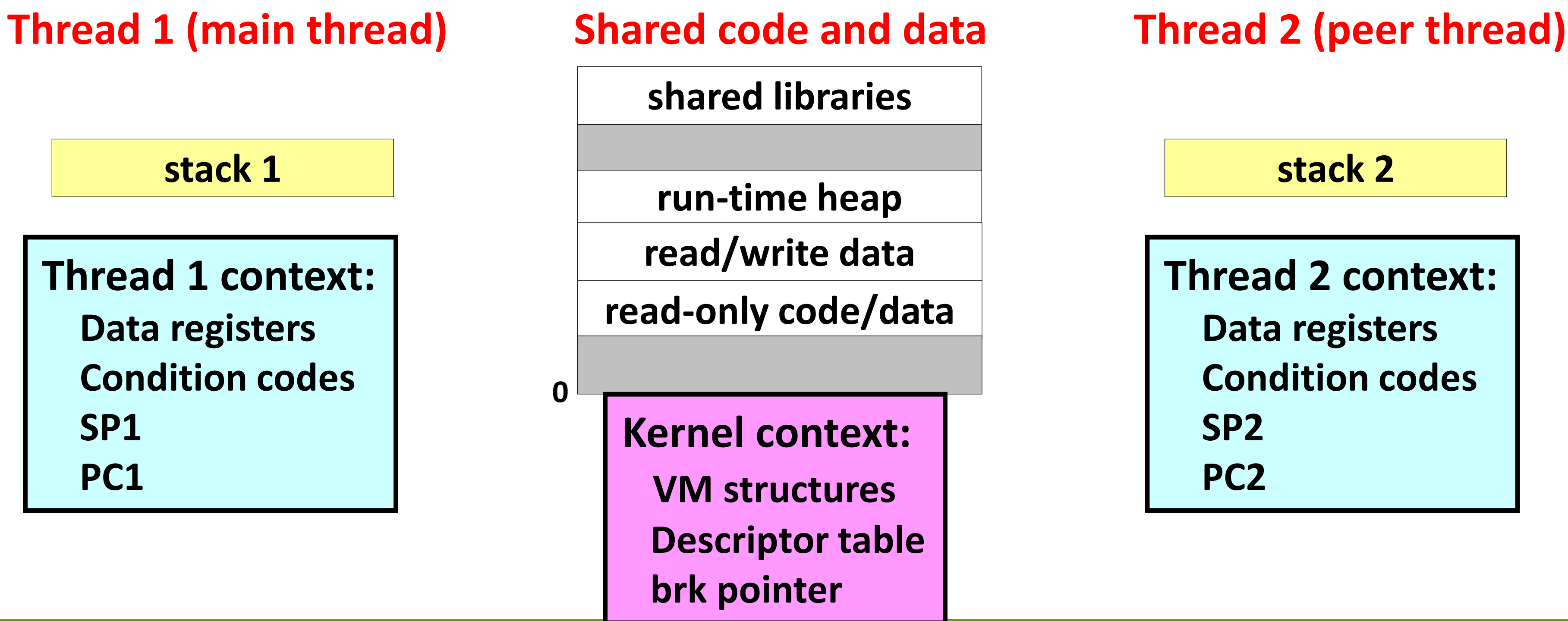
Alternate View

- Process = thread context + kernel context + address space



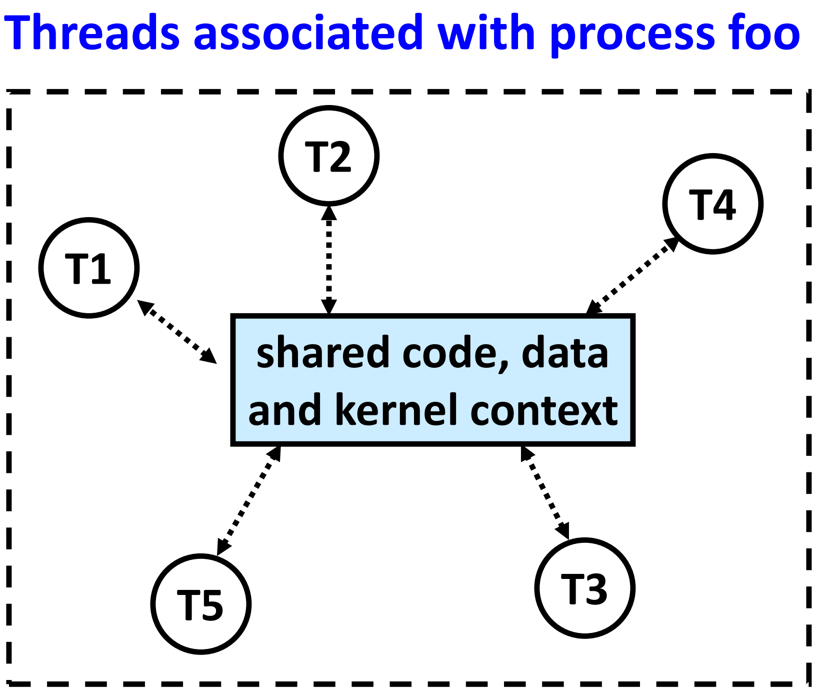
A Process with Multiple Threads

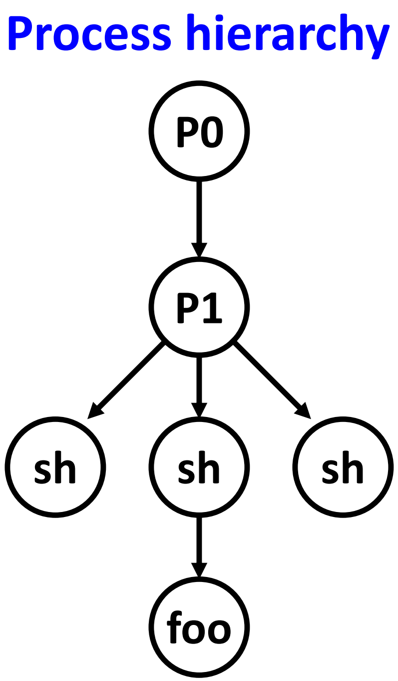
- **Multiple threads can be associated with a process.**
  - Each thread has its own logical control flow (sequence of PC values)
  - Each thread shares the same code, data, and kernel context
  - Each thread has its own thread id (TID)



Logical View of Threads

- **Threads associated with a process form a pool of peers**
  - Unlike processes which form a tree hierarchy





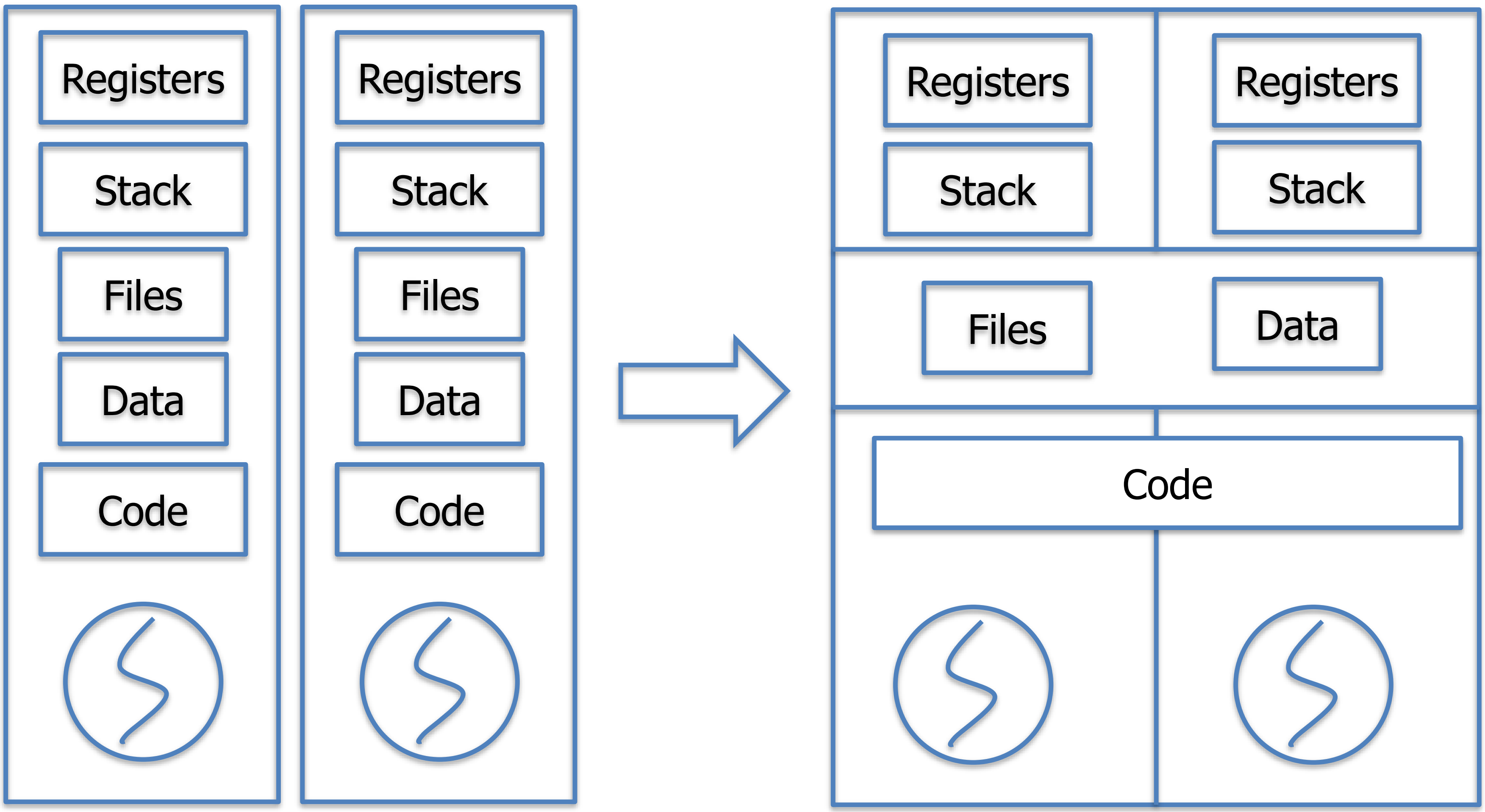
Threads vs. Processes

- **How threads and processes are similar**
  - Each has its own logical control flow.
  - Each can run concurrently.
  - Each is context switched.
- **How threads and processes are different**
  - Threads share code and data, processes (typically) do not.
  - Threads are somewhat less expensive than processes.
    - Linux 2.4 Kernel, 512MB RAM, 2 CPUs

      -> 1,811 forks()/second

      -> 227,611 threads/second (125x faster)

Threads vs. Processes



2 processes

2 threads, 1 process

Pthreads Interface

- **POSIX Threads Interface**
  - Creating and reaping threads
    - **pthread_create()**
    - **pthread_join()**
  - Determining your thread ID
    - **pthread_self()**
  - Terminating threads
    - **pthread_cancel()**
    - **pthread_exit()**
    - **exit** (terminates all threads), **return** (terminates current thread)
  - Synchronizing access to shared variables
    - **pthread_mutex_init()**
    - **pthread_mutex_[un]lock()**
    - **pthread_cond_init()**
    - **pthread_cond_[timed]wait()**
    - **pthread_cond_signal()**, etc.

"hello, world" Program (1)

```
/*
 * hello.c - Pthreads "hello, world" program
 */
#include "pthread.h"

void *thread(void *vargp);

int main() {
  pthread_t tid;

  pthread_create(&tid, NULL, thread, NULL);
  pthread_join(tid, NULL);
  exit(0);
}

/* thread routine */
void *thread(void *vargp) {
  printf("Hello, world!\n");
  return NULL;
}
```

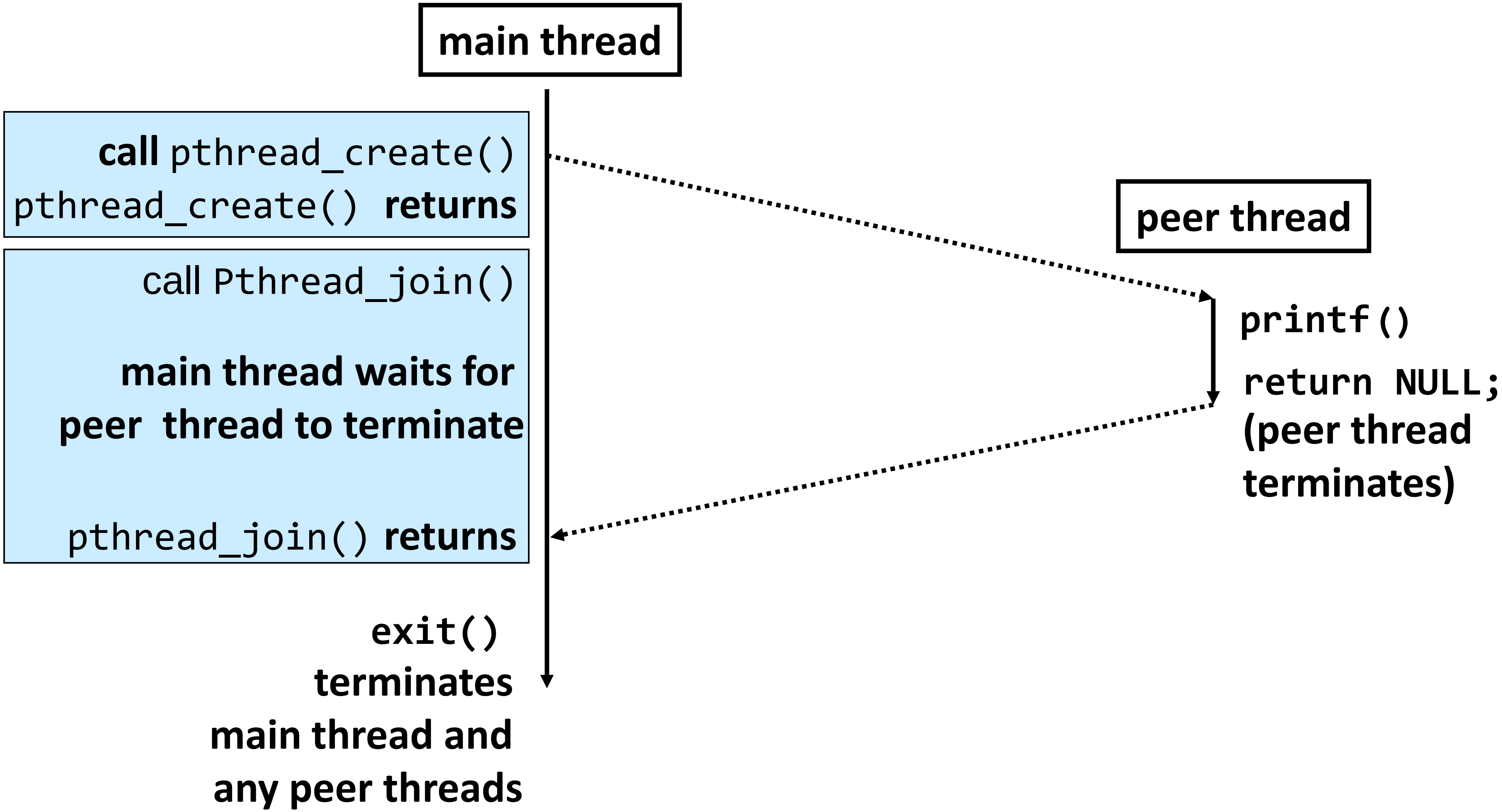
Thread attributes (usually NULL)

Thread arguments (void *p)

return value (void **p)

"hello, world" Program (2)

- Execution of threaded "hello, world"



Echo Server: Thread-based

```c
int main (int argc, char *argv[])
{
   int *connfdp;
   pthread_t tid;
   . . .

   while (1) {
      connfdp = (int *)
                malloc(sizeof(int));
      *connfdp = accept (listenfd,
          (struct sockaddr *)&caddr,
          &caddrlen));

      pthread_create(&tid, NULL,
          thread_main, connfdp);
   }
}
```

```c
void *thread_main(void *arg)
{
  int n;
  char buf[MAXLINE];

  int connfd = *((int *)arg);
  pthread_detach(pthread_self());
  free(arg);

  while((n = read(connfd, buf,
             MAXLINE)) > 0)
    write(connfd, buf, n);

  close(connfd);
  return NULL;
}
```

Implementation Issues (1)

- **Must run "detached" to avoid memory leak.**
  - At any point in time, a thread is either joinable or detached.
  - Joinable thread can be reaped and killed by other threads
    - Must be reaped (with **pthread_join**()) to free memory resources.
  - Detached thread cannot be reaped or killed by other threads.
    - Resources are automatically reaped on termination.
    - Exit state and return value are not saved.
  - Default state is joinable.
    - Use **pthread_detach(pthread_self())** to make detached.

Implementation Issues (2)

- **Must be careful to avoid unintended sharing**
  - For example, what happens if we pass the address connfd to the thread routine?

```
int connfd;
. . .
pthread_create(&tid, NULL, thread_main, &connfd);
. . .
```

- **All functions called by a thread must be thread-safe.**
  - A function is said to be thread-safe or reentrant, when the function may be called by more than one thread at a time without requiring any other action on the caller's part.

Thread-based Designs

- **Pros**
  - Easy to share data structures between threads.
    - e.g., logging information, file cache, etc.
  - Threads are more efficient than processes.
- **Cons**
  - Unintentional sharing can introduce subtle and hard-to-reproduce errors!
    - The ease with which data can be shared is both the greatest strength and the greatest weakness of threads.

Thread Safety (1)

- **Thread-safe**
  - Functions called from a thread must be thread-safe.
  - We identify four (non-disjoint) classes of thread-unsafe functions:
    - Class 1: Failing to protect shared variables
    - Class 2: Relying on persistent state across invocations
    - Class 3: Returning a pointer to a static variable
    - Class 4: Calling thread-unsafe functions

Thread Safety (2)

- **Class 1: Failing to protect shared variables.**
  - Fix: Use mutex operations.
  - Issue: Synchronization operations will slow down code.

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
int cnt = 0;

/* Thread routine */
void *count(void *arg) {
    int i;

    for (i=0; i<NITERS; i++) {
        pthread_mutex_lock (&lock);
        cnt++;
        pthread_mutex_unlock (&lock);
    }
    return NULL;
}
```

Thread Safety (3)

- **Class 2: Relying on persistent state across multiple function invocations.**
  - Random number generator relies on static state
  - Fix: Rewrite function so that caller passes in all necessary state.

```
/* rand - return pseudo-random integer on 0..32767 */
int rand(void) {
    static unsigned int next = 1;
    next = next*1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}
/* srand - set seed for rand() */
void srand(unsigned int seed) {
    next = seed;
}
```

Thread Safety (4)

- **Class 3: Returning a ptr to a static variable.**
- **Fixes:**
  1. Rewrite code so caller passes pointer to `struct`.
     - Issue: Requires changes in caller and callee.
  2. *Lock-and-copy*
     - Issue: Requires only simple changes in caller (and none in callee)
       - » However, caller must free memory.

```
struct hostent
*gethostbyname(char *name){
  static struct hostent h;
  <contact DNS and fill in h>
  return &h;
}
```

```
hostp = malloc(...));
gethostbyname_r(name, hostp);
```

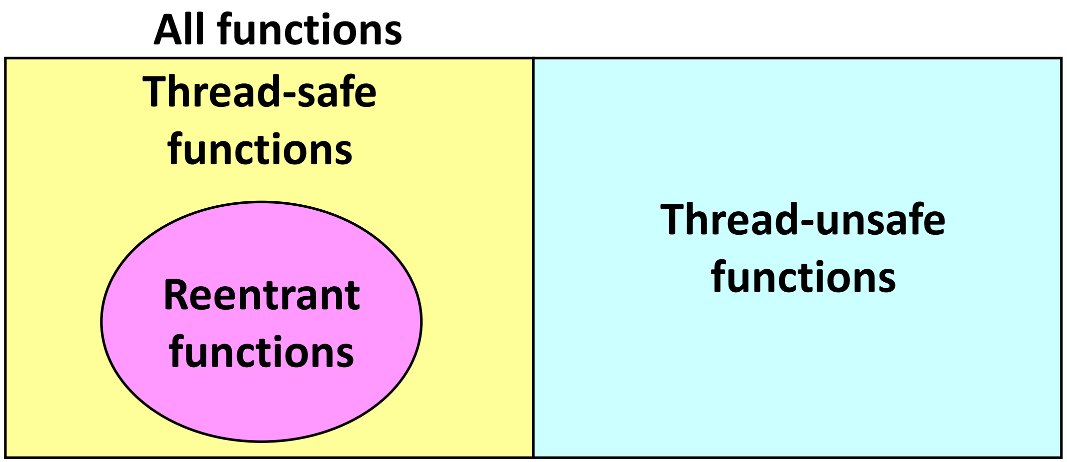
```
struct hostent
*gethostbyname_ts(char *name)
{
  struct hostent *unshared
                       = malloc(...);
  pthread_mutex_lock(&lock); /* lock */
  shared = gethostbyname(name);
  *unshared = *shared;    /* copy */
  pthread_mutex_unlock(&lock);
  return q;
}
```

Thread Safety (5)

- **Class 4: Calling thread-unsafe functions.**
  - Calling one thread-unsafe function makes an entire function thread-unsafe.
  - Fix: Modify the function so it calls only thread-safe functions

Reentrant Functions

- **A function is *reentrant* iff it accesses NO shared variables when called from multiple threads.**
  - Reentrant functions are a proper subset of the set of thread-safe functions.



    – NOTE: The fixes to Class 2 and 3 thread-unsafe functions require modifying the function to make it reentrant.

Thread-Safe Library

- **All functions in the Standard C Library (at the back of your K&R text) are thread-safe.**
  - Examples: `malloc, free, printf, scanf`
- **Most Unix system calls are thread-safe, with a few exceptions:**

Thread-unsafe function	Class	Reentrant version
asctime	3	asctime_r
ctime	3	ctime_r
gethostbyaddr	3	gethostbyaddr_r
gethostbyname	3	gethostbyname_r
inet_ntoa	3	(none)
localtime	3	localtime_r
rand	2	rand_r

Concurrent Programming

Examples

Example

- **Make chatting server**
  - N client + 1 server
  - A thread from the server reads messages from 1 client, and sends messages to all other clients
  - Must wait when client is full
- **Use pthread_mutex**
  - When accessing shared data structure among threads
  - When condition variable is needed