

Concurrent Programming

Dong-kun Shin
Embedded Software Laboratory
Sungkyunkwan University
<http://nyx.skku.ac.kr>

Echo Server Revisited

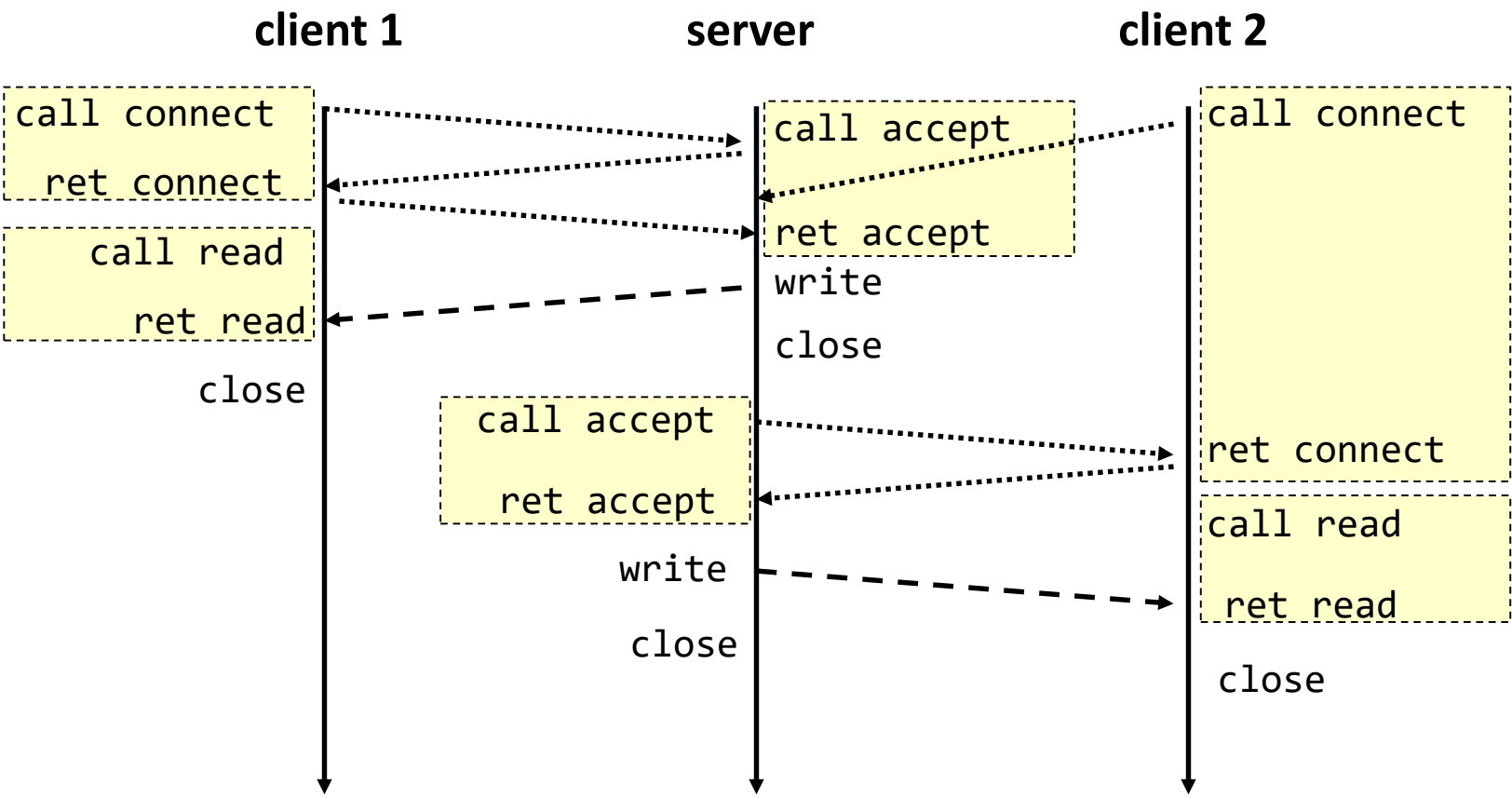
```
int main (int argc, char *argv[]) {
    ...
    listenfd = socket(AF_INET, SOCK_STREAM, 0);

    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, &crlen);
        while ((n = read(connfd, buf, MAXLINE)) > 0) {
            printf ("got %d bytes from client.\n", n);
            write(connfd, buf, n);
        }
        close(connfd);
    }
}
```

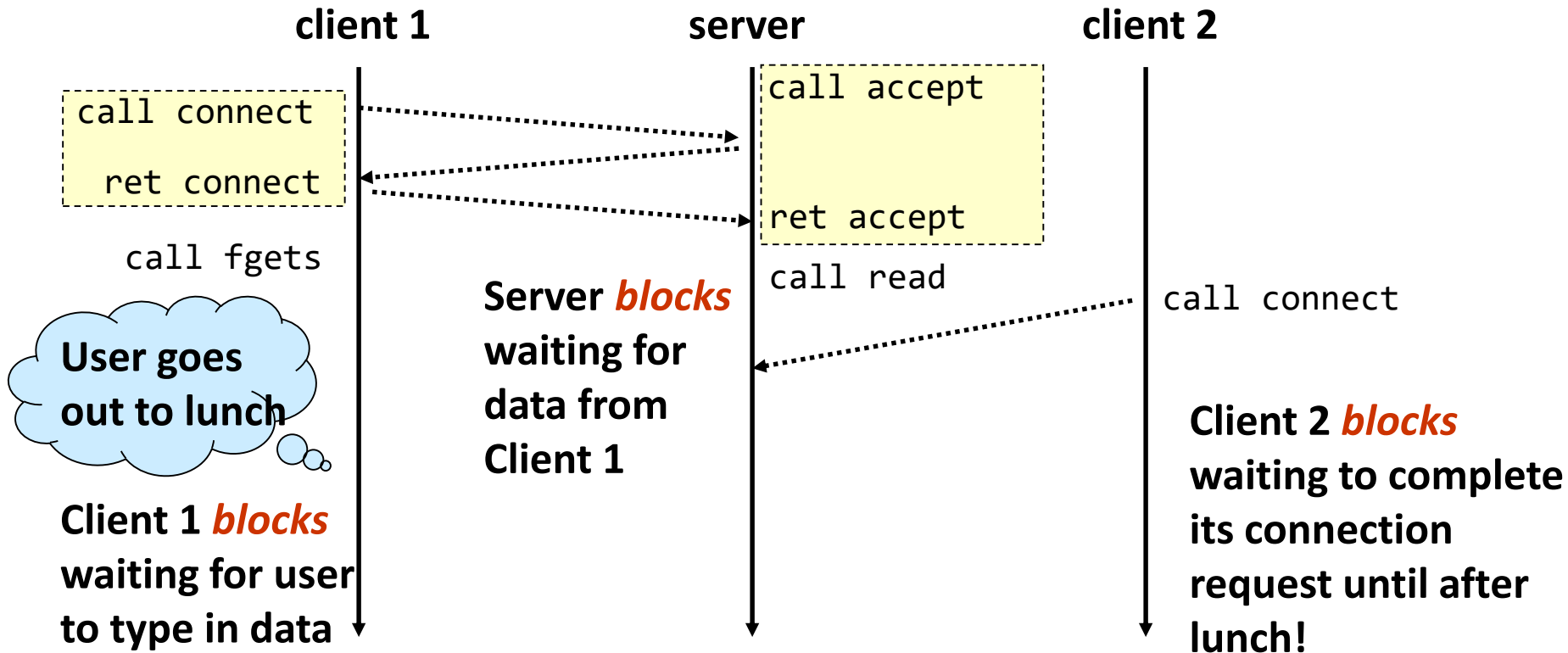
Iterative Servers (1)

- 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.

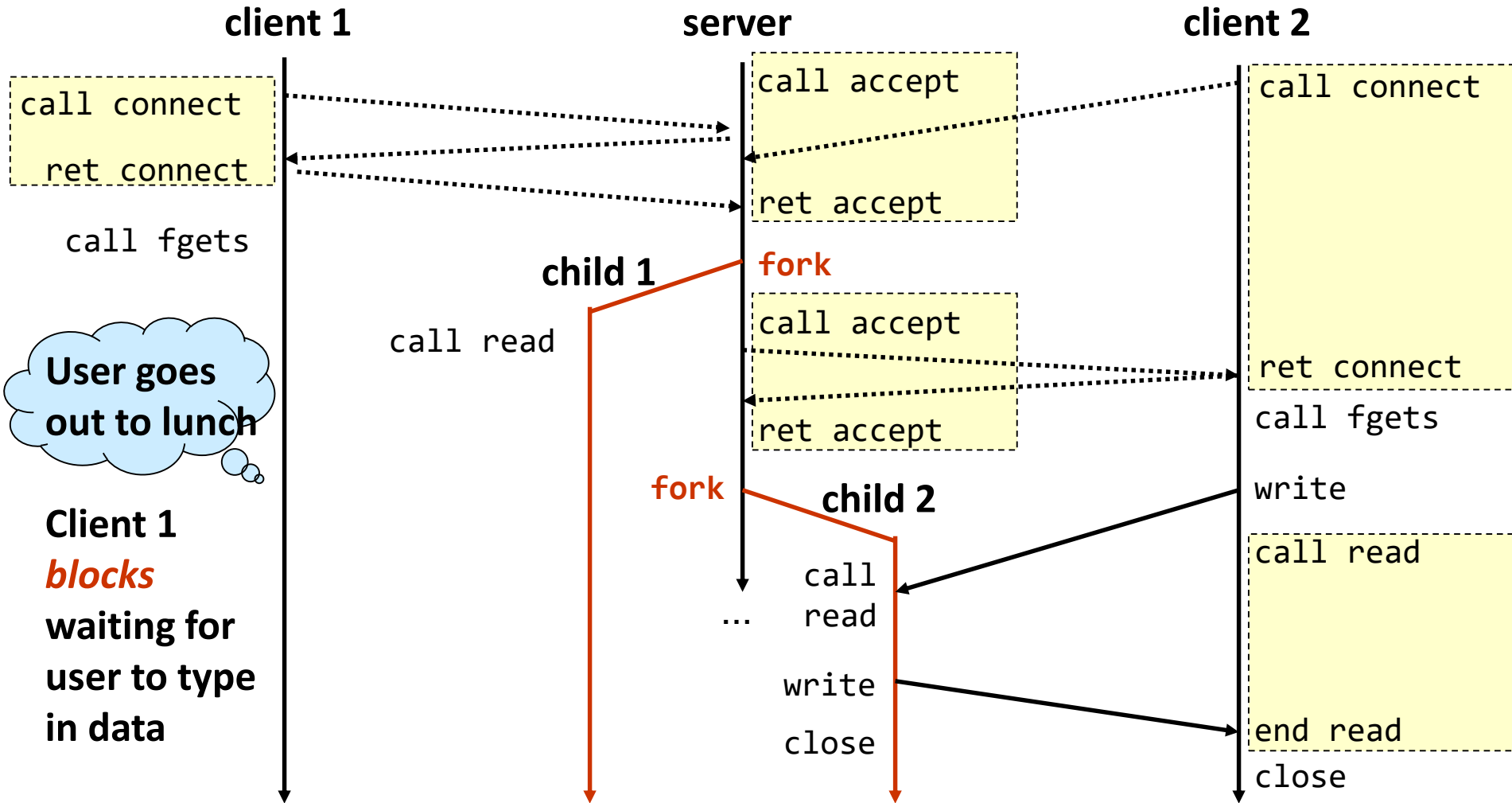
Creating Concurrent Flows

- 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

Process-based

Process-based Servers



Echo Server

- Iterative version

```
int main (int argc, char *argv[])
{
    . . .

    while (1) {
        connfd = accept (listenfd, (struct sockaddr *)&caddr,
                        &caddrlen));

        while ((n = read(connfd, buf, MAXLINE)) > 0) {
            printf ("got %d bytes from client.\n", n);
            write(connfd, buf, n);
        }

        close(connfd);
    }
}
```


Echo Server: Process-based

```
int main (int argc, char *argv[])
{
    . . .
    signal (SIGCHLD, handler);

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

```
void handler(int sig) {
    pid_t pid;
    int stat;
    while ((pid = waitpid(-1, &stat,
                        WNOHANG)) > 0);
    return;
}
```

Implementation Issues

- Server must reap zombie children
 - to avoid fatal memory leak
- Server must close its copy of `connfd`.
 - Kernel keeps reference for each socket.
 - After **`fork()`**, **`refcnt(connfd) = 2`**
 - Connection will not be closed until **`refcnt(connfd) = 0`**

Process-based Designs

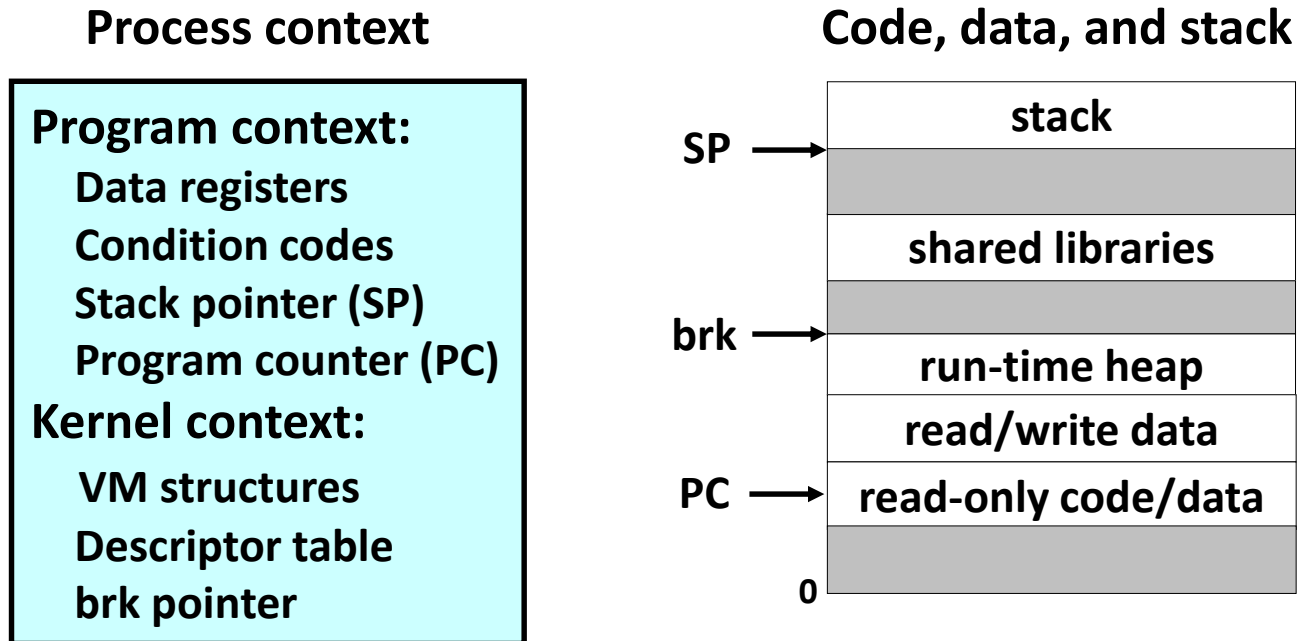
- Pros
 - Handles multiple connections concurrently.
 - Clean sharing model.
 - Descriptors (no), file tables (yes), global variables (no)
 - Simple and straightforward.
- Cons
 - Additional overhead for process control.
 - Process creation and termination
 - Process switching
 - Nontrivial to share data between processes.
 - Requires IPC (InterProcess Communication) mechanisms: FIFO's, System V shared memory and semaphores

Concurrent Programming

Thread-based

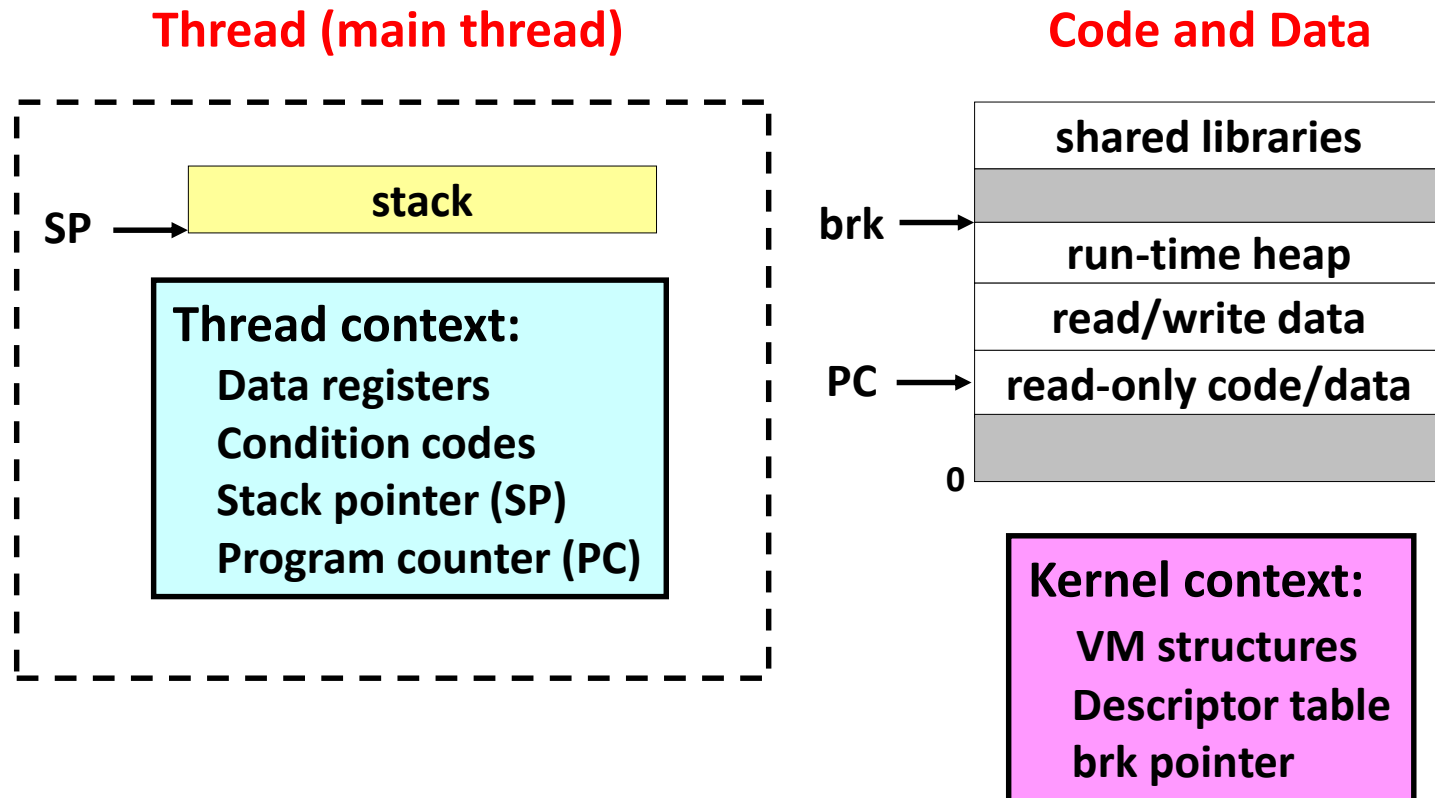
Traditional View

- Process = process context + address space



Alternate View

- Process = thread context + kernel context + address space



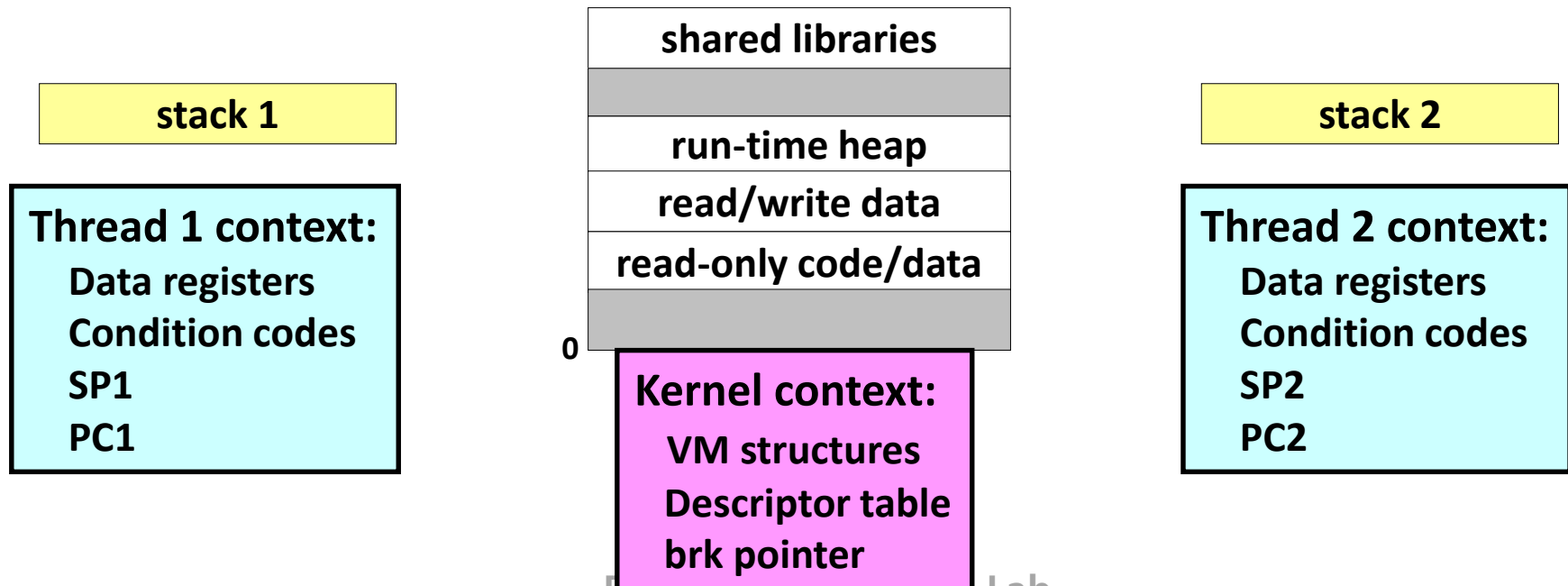
A Process with Multiple Threads

- 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)

Thread 1 (main thread)

Shared code and data

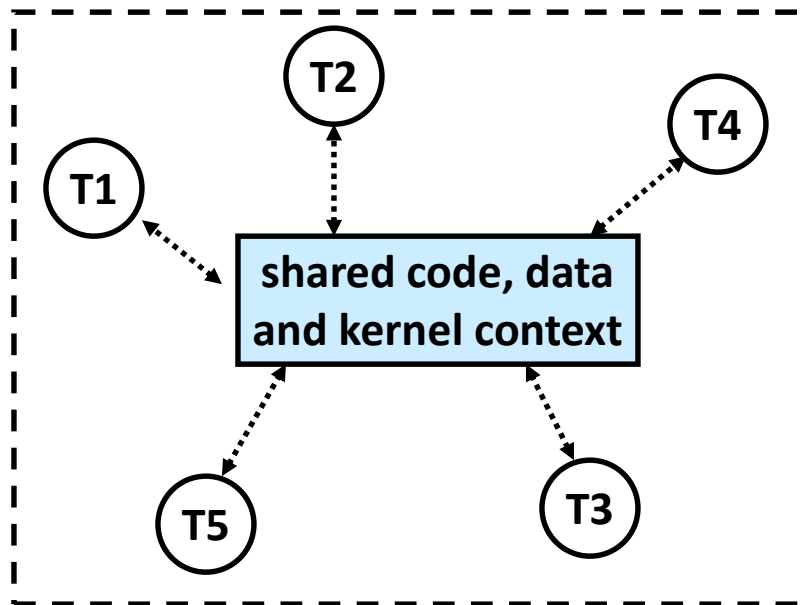
Thread 2 (peer thread)



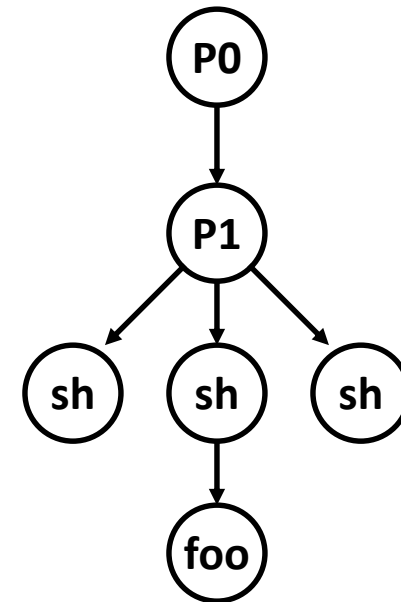
Logical View of Threads

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

Threads associated with process foo



Process 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)

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;
}
```

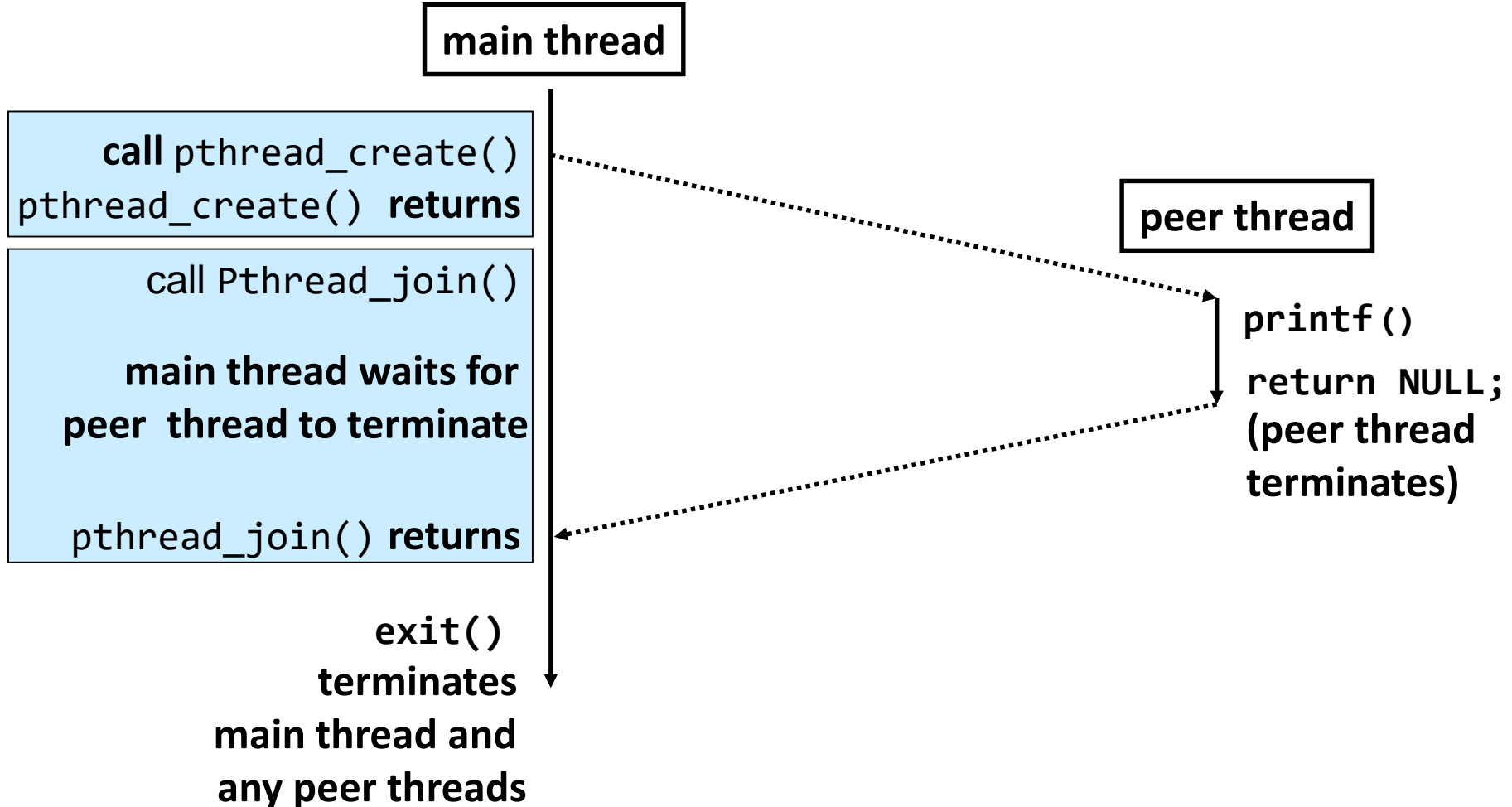
*Thread attributes
(usually NULL)*

*Thread arguments
(void *p)*

*return value
(void **p)*

"hello, world" Program (2)

- Execution of threaded "hello, world"



Echo Server: Thread-based

```
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);
    }
}
```

```
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.

Exercise

- Concurrent program 성능 비교

- 3가지 program의 실행 시간 측정

- Iterative programming
- Process-based programming
- Thread-based programming

- *calc* function을 10번 수행

- 10 fork
- 10 pthread

- Note

- `while(waitpid(-1, &stat, 0) > 0);`
- `gcc your_code.c -lm -pthread -o a.out`

```
double calc(void)
{
    int i;
    double sum = 0;

    for(i = 1; i <= 1000000; i++)
        sum += pow(log(i), 0.5);
    return sum;
}
```

Exercise-pthread code

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <pthread.h>

pthread_t tid[10];
double sum[10];

double calc(void)
{
    int i;
    double sum = 0;

    for(i = 1; i <= 1000000; i++)
        sum += pow(log(i), 0.5);
    return sum;
}

void *thread_main(int* pi)
{
    int i = *pi;
    free(pi);
    sum[i] = calc();
}
```

```
int main(void)
{
    int i;
    int *pi;
    double total = 0;

    for(i = 0; i < 10; i++) {
        pi = (int*)malloc(sizeof(int));
        *pi = i;
        pthread_create(&tid[i], NULL, thread_main, pi);
    }

    for(i = 0; i < 10; i++) {
        pthread_join(tid[i], NULL);
        total += sum[i];
    }

    printf("Total: %lf\n", total);
    return 0;
}
```