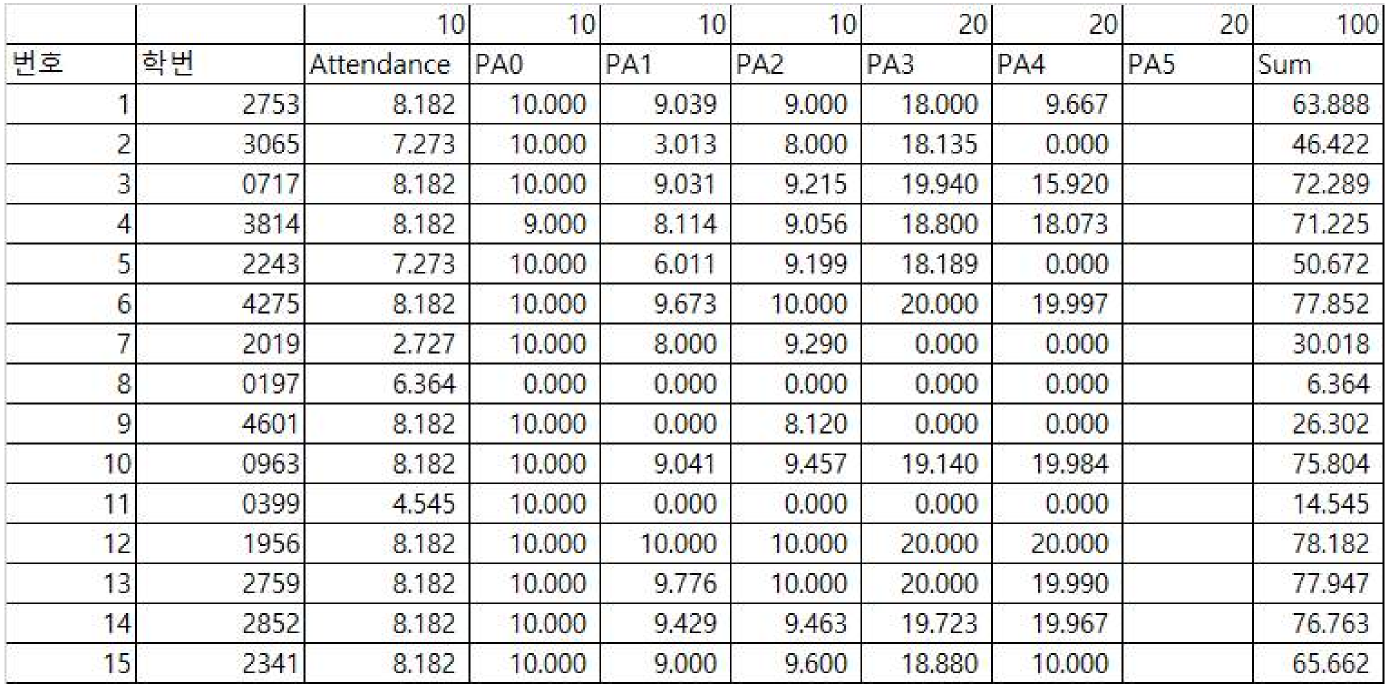
Announcement

- **'Total Score' is updated**
  - Please check it

Will be changed
↓



		10	10	10	10	20	20	20	100
번호	학번	Attendance	PA0	PA1	PA2	PA3	PA4	PA5	Sum
1	2753	8.182	10.000	9.039	9.000	18.000	9.667		63.888
2	3065	7.273	10.000	3.013	8.000	18.135	0.000		46.422
3	0717	8.182	10.000	9.031	9.215	19.940	15.920		72.289
4	3814	8.182	9.000	8.114	9.056	18.800	18.073		71.225
5	2243	7.273	10.000	6.011	9.199	18.189	0.000		50.672
6	4275	8.182	10.000	9.673	10.000	20.000	19.997		77.852
7	2019	2.727	10.000	8.000	9.290	0.000	0.000		30.018
8	0197	6.364	0.000	0.000	0.000	0.000	0.000		6.364
9	4601	8.182	10.000	0.000	8.120	0.000	0.000		26.302
10	0963	8.182	10.000	9.041	9.457	19.140	19.984		75.804
11	0399	4.545	10.000	0.000	0.000	0.000	0.000		14.545
12	1956	8.182	10.000	10.000	10.000	20.000	20.000		78.182
13	2759	8.182	10.000	9.776	10.000	20.000	19.990		77.947
14	2852	8.182	10.000	9.429	9.463	19.723	19.967		76.763
15	2341	8.182	10.000	9.000	9.600	18.880	10.000		65.662

- **The score of PA4 will be changed!**

Concurrent Programming

Prof. Jin-Soo Kim(jinsookim@skku.edu)
TA – Sanghoon Han(sanghoon.han@csl.skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu

Echo Server Revisited

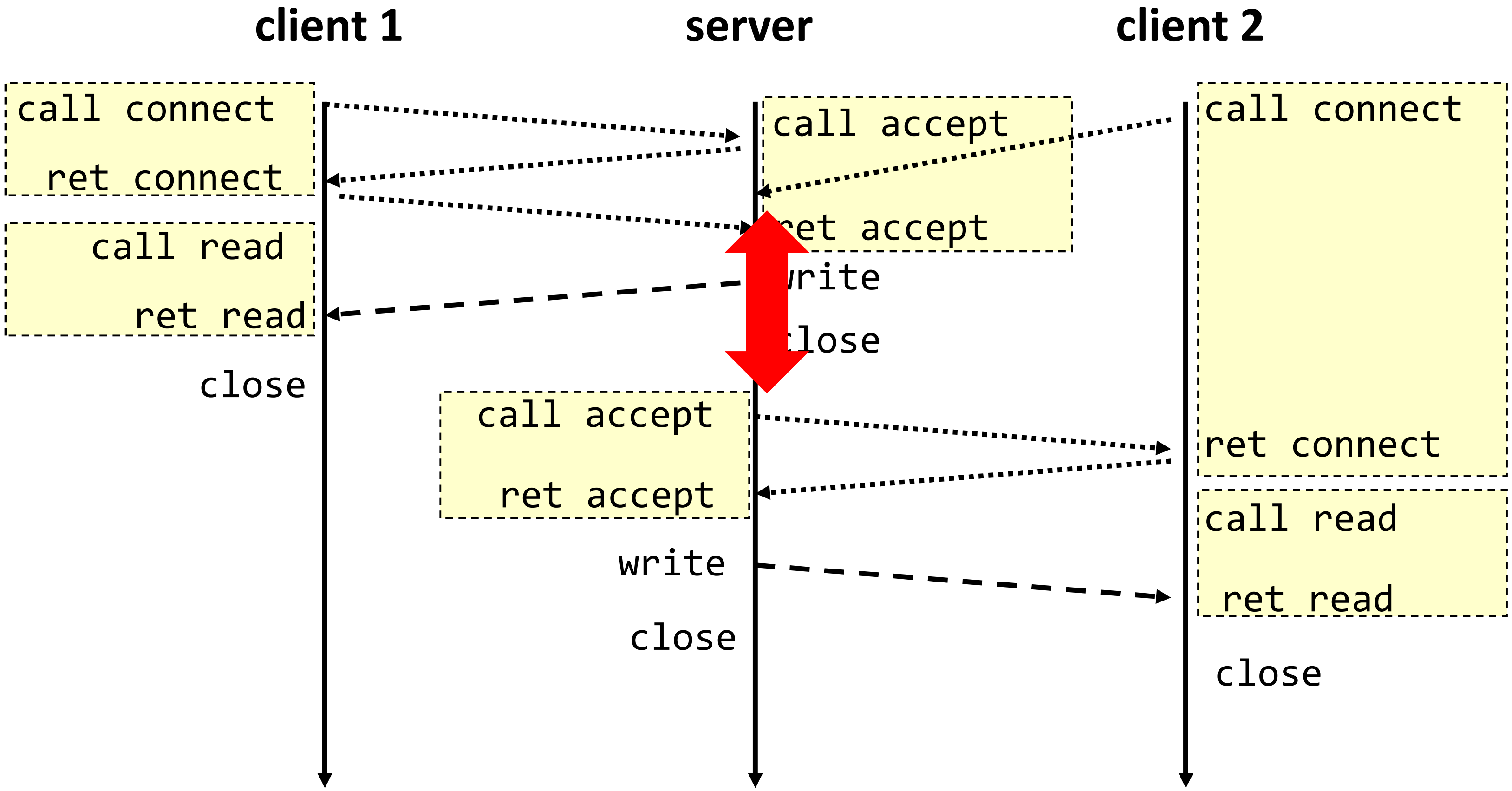
```
int main (int argc, char *argv[]) {
   ...
   listenfd = socket(AF_INET, SOCK_STREAM, 0);

   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



client 1

call connect

ret connect

call read

ret read

close

server

call accept

ret accept

write

close

call accept

ret accept

write

close

client 2

call connect

ret connect

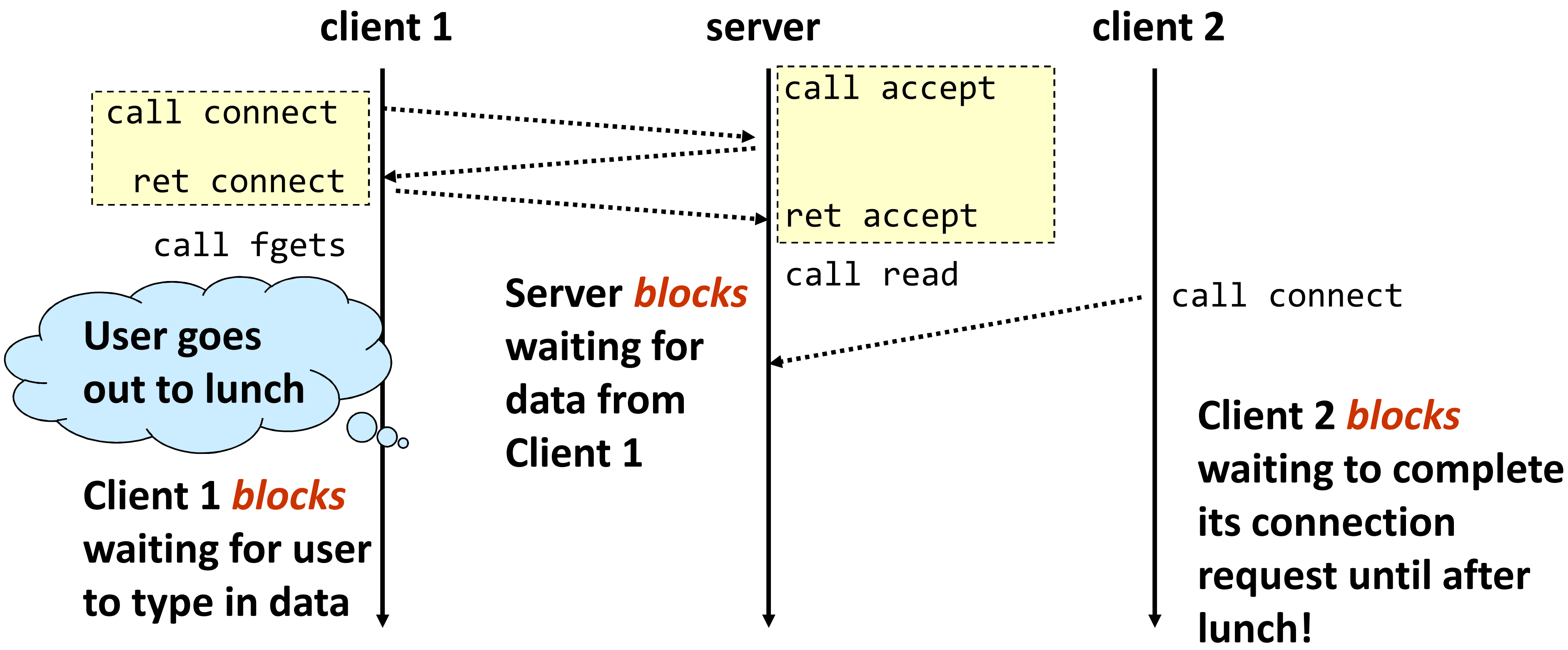
call read

ret read

close

Iterative Servers (2)

- **Fundamental flaw**



Server *blocks* waiting for data from Client 1

Client 1 *blocks* waiting for user to type in data

Client 2 *blocks* waiting to complete its connection request until after lunch!

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

Concurrent Programming

Thread-based

Traditional View

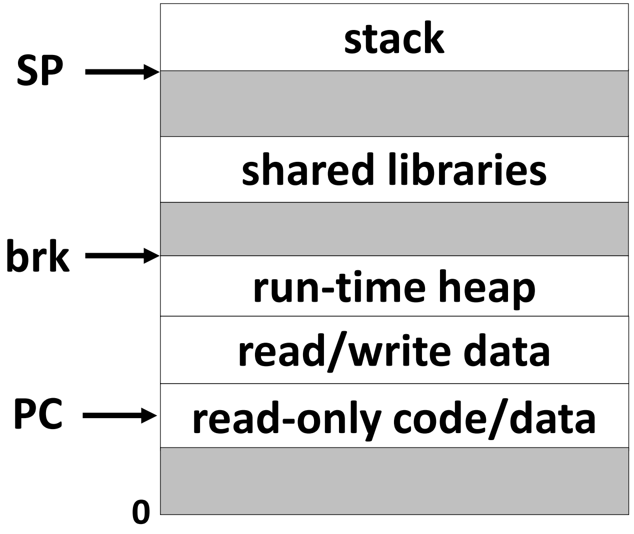
- Process = process context + address space

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



stack

SP →

shared libraries

brk →

run-time heap

read/write data

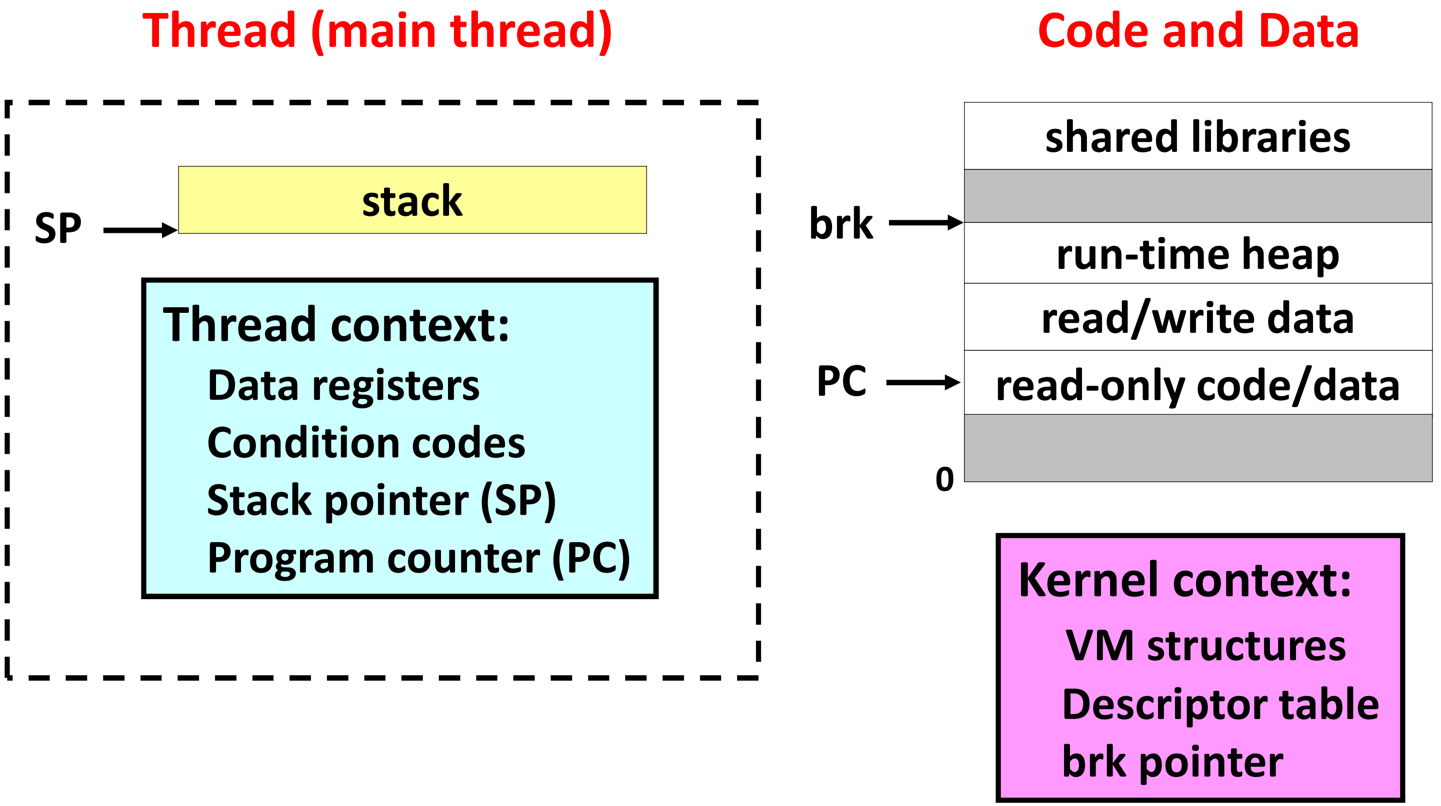
PC →

read-only code/data

0

Alternate View

- Process = thread context + kernel context + address space



Thread (main thread)

SP

stack

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

Code and Data

shared libraries

brk

run-time heap

read/write data

PC

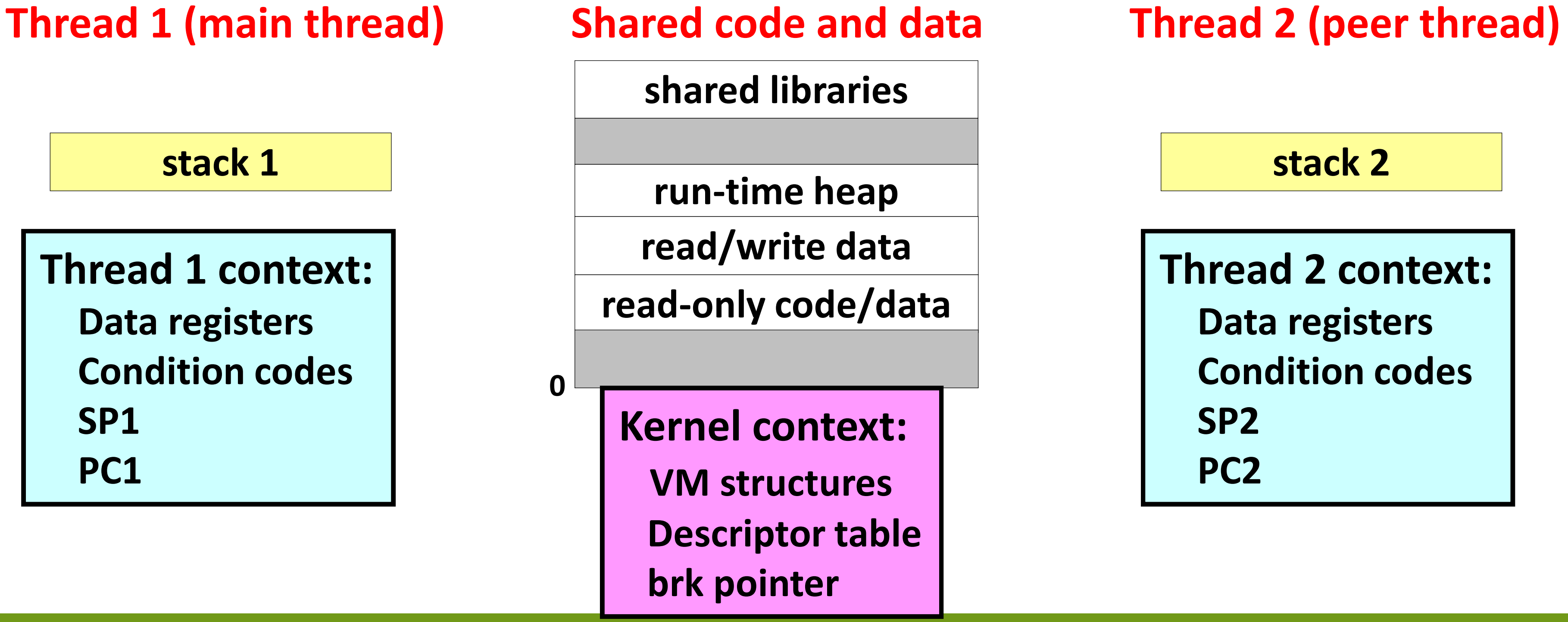
read-only code/data

0

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

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)

stack 1

Thread 1 context:
Data registers
Condition codes
SP1
PC1

Shared code and data

shared libraries

run-time heap

read/write data

read-only code/data

0

Kernel context:
VM structures
Descriptor table
brk pointer

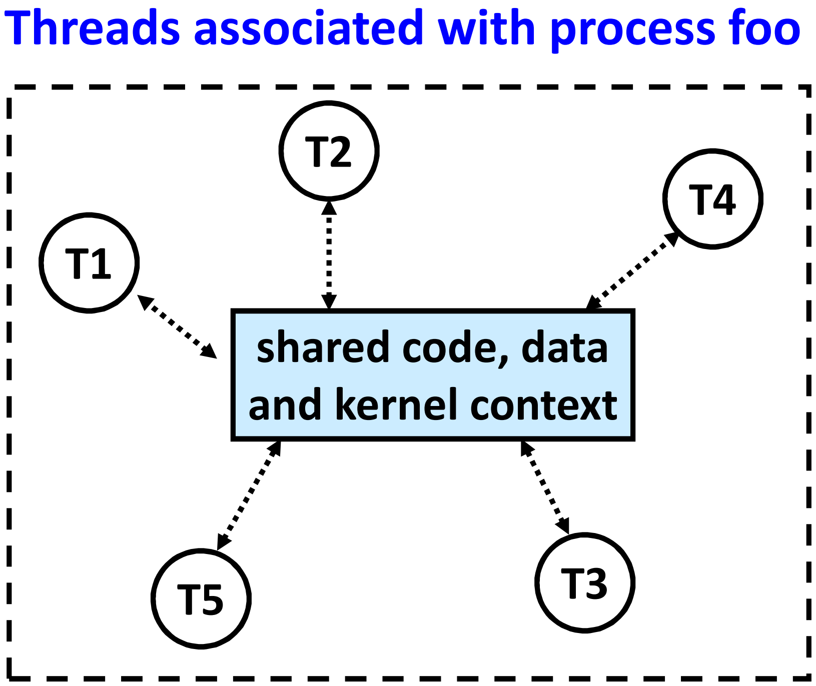
Thread 2 (peer thread)

stack 2

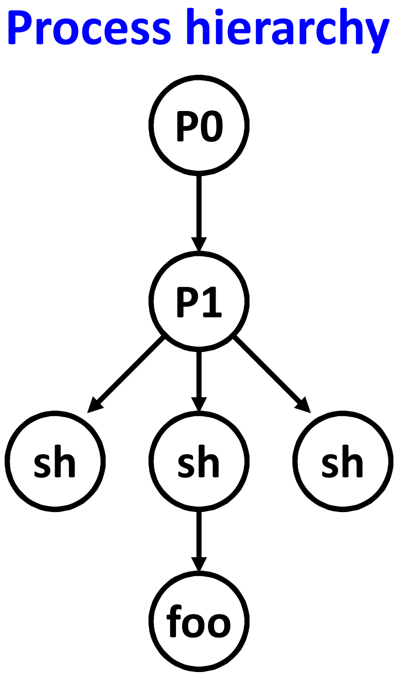
Thread 2 context:
Data registers
Condition codes
SP2
PC2

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

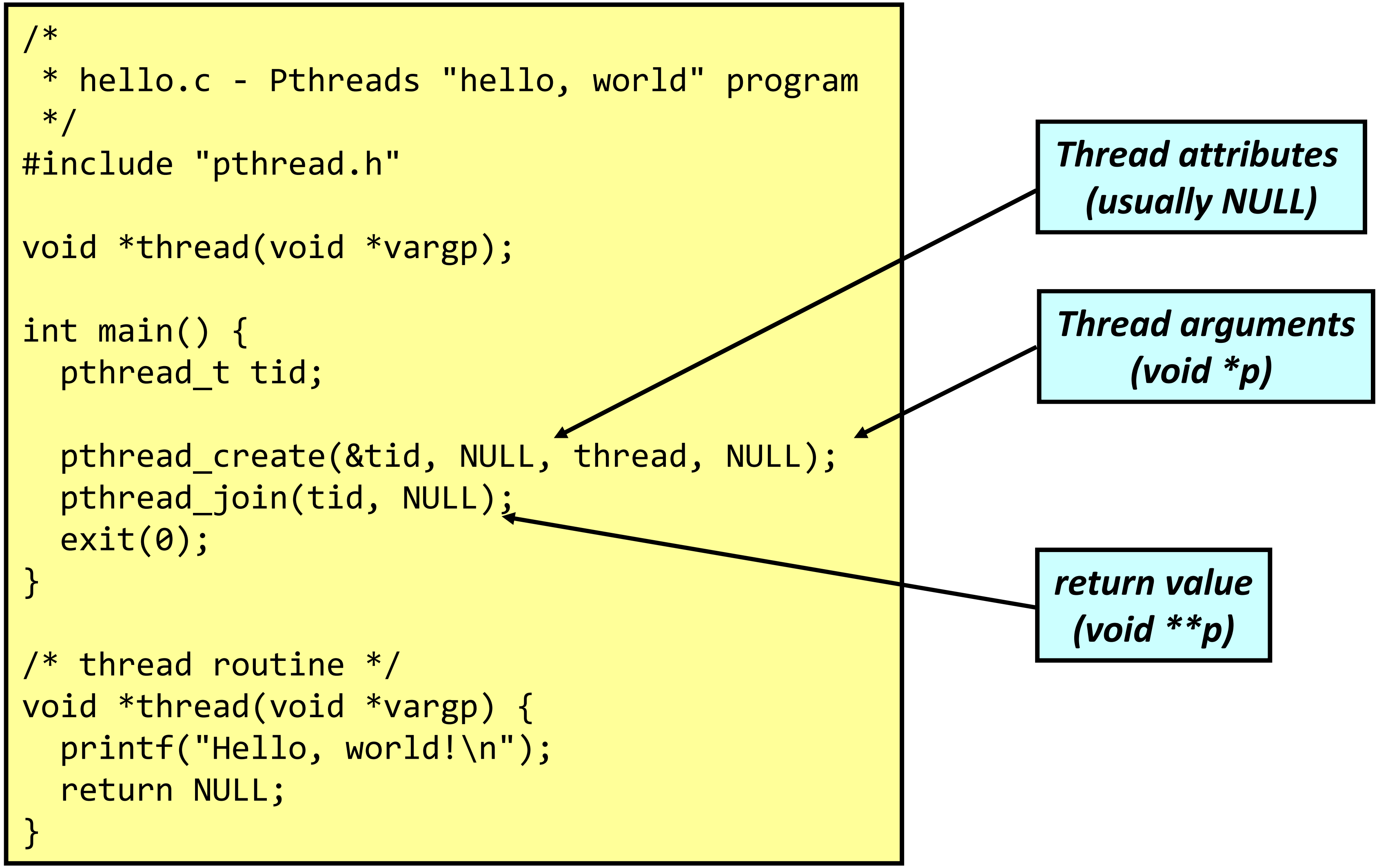
      -> 1,811 forks()/second

      -> 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;
}
```

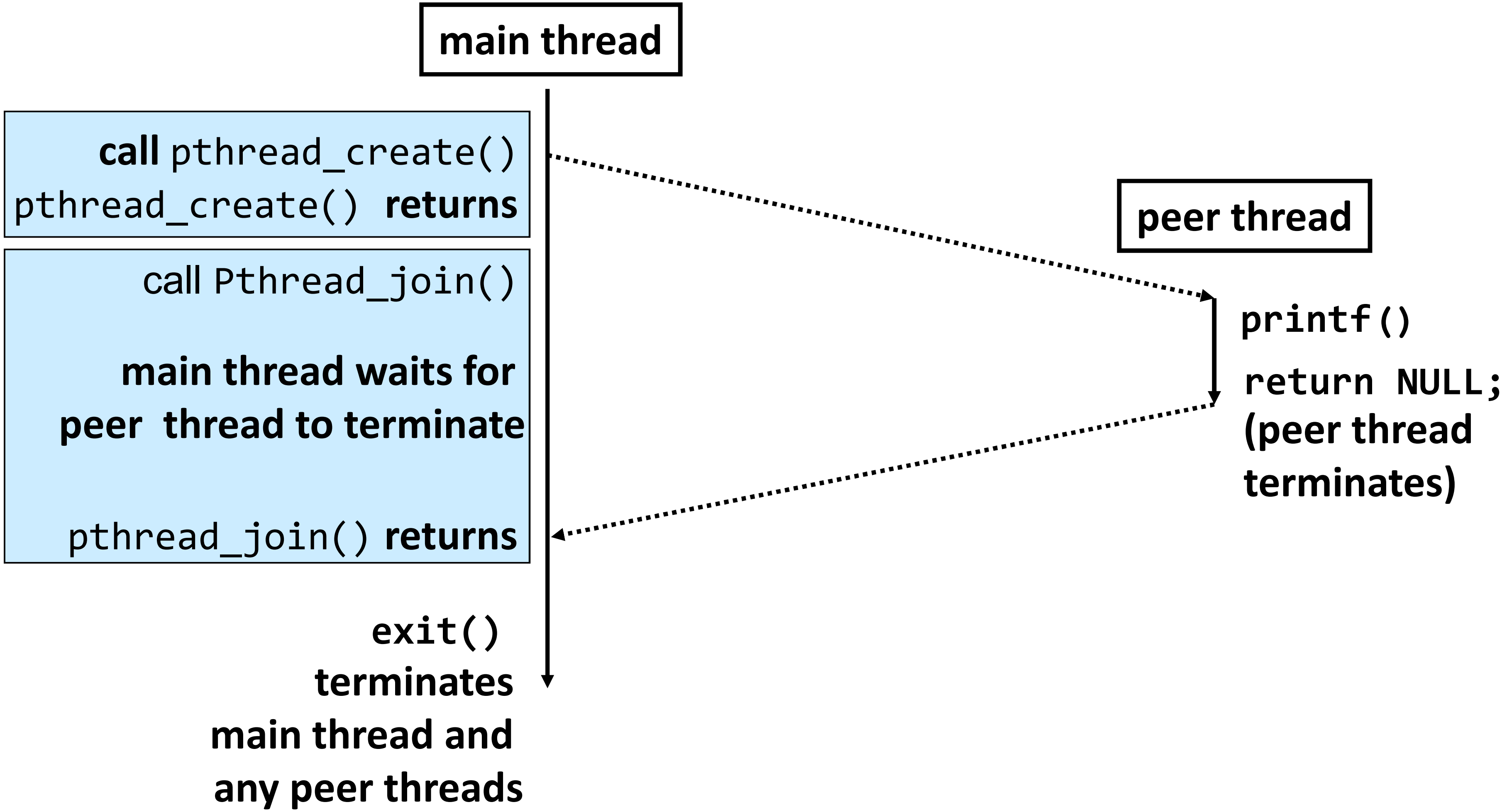
Thread attributes (usually NULL)

Thread arguments (void *p)

return value (void **p)

"hello, world" Program (2)

- **Execution of threaded "hello, world"**



main thread

call `pthread_create()`
`pthread_create()` **returns**

call `Pthread_join()`

main thread waits for peer thread to terminate

`pthread_join()` **returns**

`exit()`
terminates main thread and any peer threads

peer thread

`printf()`

`return NULL;`
(peer thread terminates)

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.

Concurrent Programming

Examples

Example 1

- **Fill skeleton code**
  - 5 Reader + 1 Writer
  - Writer updates value 1000,000 times
  - Each reader reads value 10,000,000 times

- **Use pthread_mutex**

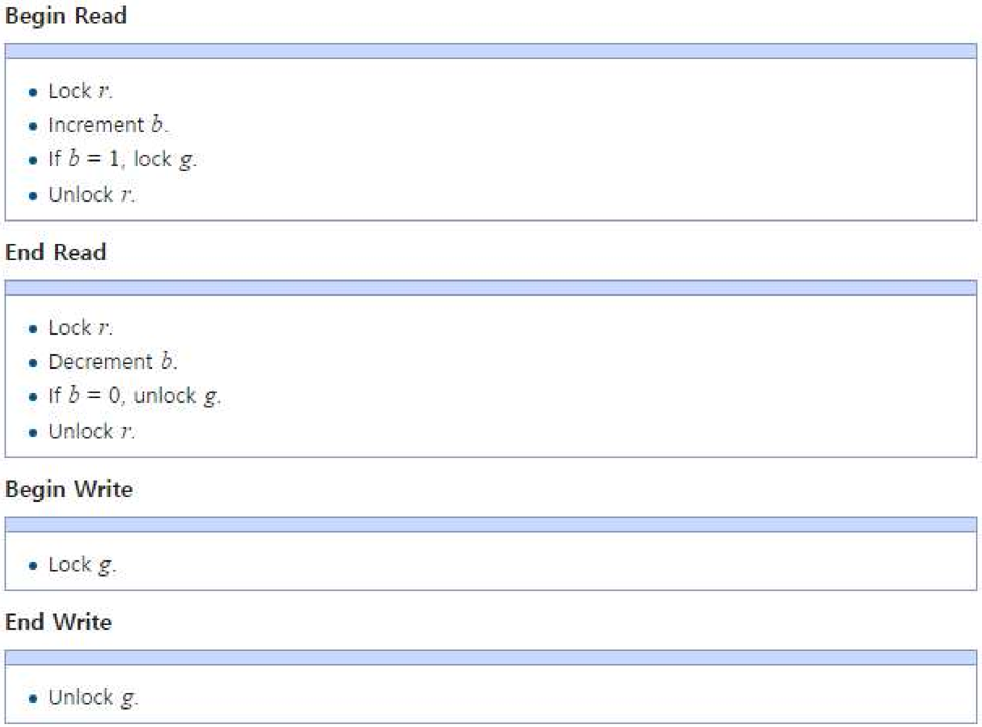
Example 2

- Implement same thing using pthread_spinlock
  - pthread_spinlock_t s
  - pthread_spin_init(&s, PTHREAD_PROCESS_PRIVATE)
  - pthread_spin_[un]lock(&s)
  - pthread_spin_destroy(&s)

- What is the difference?

Readers-Writer Lock

- **Reader blocks other reader**
  - Do we need this?
- **Implementation (with 2 lock)**



Begin Read

- Lock r.
- Increment b.
- If $b = 1$, lock g.
- Unlock r.

End Read

- Lock r.
- Decrement b.
- If $b = 0$, unlock g.
- Unlock r.

Begin Write

- Lock g.

End Write

- Unlock g.

Example 3

- **Implement same thing with readers-writer lock**