Inter-Process Communication

Disclaimer: some slides are adopted from the book authors' slides with permission

Today

- Inter-Process Communication (IPC)
  - What is it?
  - What IPC mechanisms are available?

Inter-Process Communication (IPC)

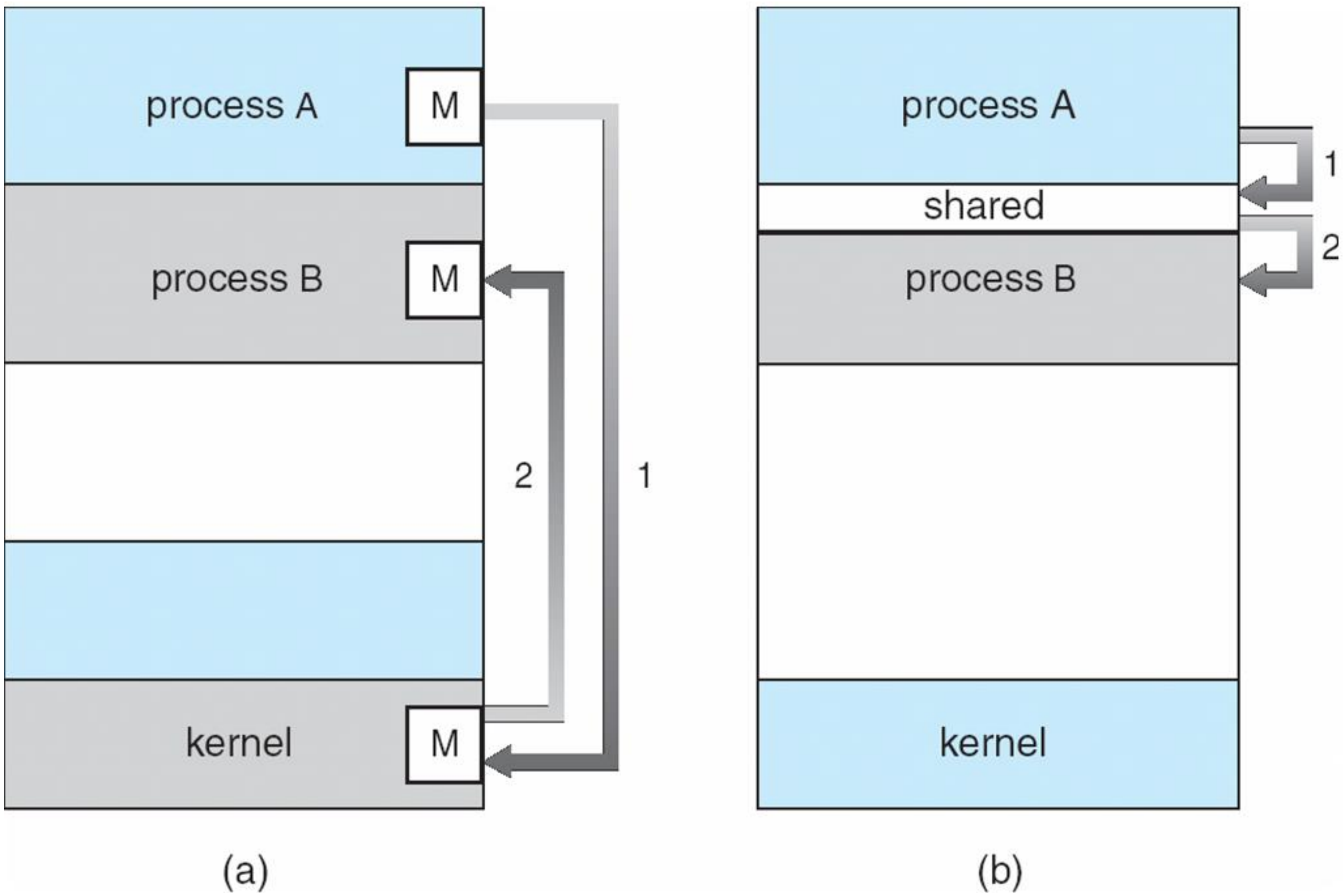
- What is it?
  - Communication among processes

- Why needed?
  - Information sharing
  - Modularity
  - Speedup

Chrome Browser

- Multi-process architecture
- Each tab is a **separate** process
  - Why?
  - How to communicate among the processes?

Models of IPC



message passing

shared memory

Models of IPC

- Shared memory
  - share a region of memory between co-operating processes
  - read or write to the shared memory region
  - ++ fast communication
  - -- synchronization is very difficult

- Message passing
  - exchange messages (send and receive)
  - typically involves data copies (to/from buffer)
  - ++ synchronization is easier
  - -- slower communication

Interprocess Communication in Unix (Linux)

- **Pipe**
- **FIFO**
- **Shared memory**
- **Socket**
- Message queue
- …

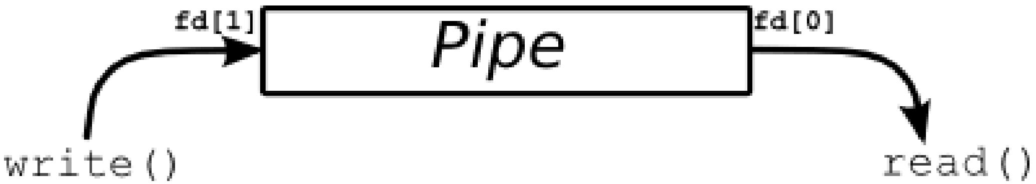
Pipes

- Most basic form of IPC on all Unix systems
  - Your shell uses this a lot (and your 1st project too)

```
ls | more
```

- Characteristics
  - Unidirectional communication
  - Processes must be in the same OS
  - Pipes exist only until the processes exist
  - Data can only be collected in FIFO order

IPC Example Using Pipes



(*) Img. source: http://beej.us/guide/bgipc/output/html/multipage/pipes.html

```
main()
{
  char *s, buf[1024];
  int fds[2];
  s  = "Hello World\n";

  /* create a pipe */
  pipe(fds);

  /* create a new process using fork */
  if (fork() == 0) {

    /* child process. All file descriptors, including
       pipe are inherited, and copied.*/
    write(fds[1], s, strlen(s));
    exit(0);
  }

  /* parent process */
  read(fds[0], buf, strlen(s));
  write(1, buf, strlen(s));
}
```

Pipes in Shells

- Example: $ ls| more
  - The output of ‘ls’ becomes the input of ‘more’
- How does the shell realize this command?
  - Create a pipe
  - Create a process to run *ls*
  - Create a process to run *more*
  - The standard output of the process to run *ls* is redirected to a pipe streaming to the process to run *more*
  - The standard input of the process to run *more* is redirected to be the pipe from the process running *ls*

KU THE UNIVERSITY OF KANSAS

FIFO (Named Pipe)

- Pipe with a name!
- More powerful than anonymous pipes
  - No parent-sibling relationship required
  - Allow bidirectional communication
  - FIFOs exists even after creating process is terminated
- Characteristics of FIFOs
  - Appear as typical files
  - Communicating process must reside on the same machine

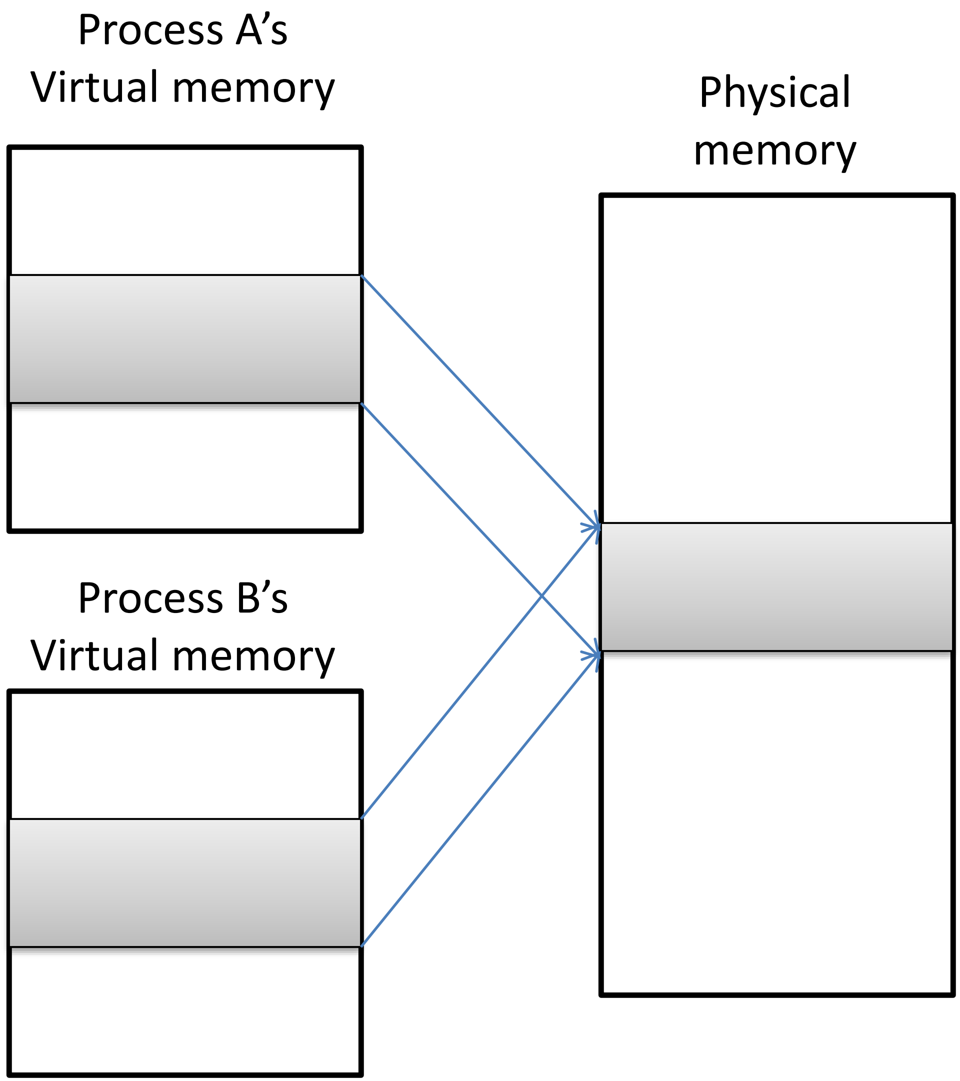
Example: Producer

```
main()
{
  char str[MAX_LENGTH];
  int num, fd;

  mkfifo(FIFO_NAME, 0666); // create FIFO file
  fd = open(FIFO_NAME, O_WRONLY); // open FIFO for writing

  printf("Enter text to write in the FIFO file: ");
  fgets(str, MAX_LENGTH, stdin);
  while(!(feof(stdin))){
    if ((num = write(fd, str, strlen(str))) == -1)
      perror("write");
    else
      printf("producer: wrote %d bytes\n", num);
    fgets(str, MAX_LENGTH, stdin);
  }
}
```

Example: Consumer

```
main()
{
  char str[MAX_LENGTH];
  int num, fd;

  mkfifo(FIFO_NAME, 0666); // make fifo, if not already present
  fd = open(FIFO_NAME, O_RDONLY); // open fifo for reading

  do{
    if((num = read(fd, str, MAX_LENGTH)) == -1)
      perror("read");
    else{
      str[num] = '\0';
      printf("consumer: read %d bytes\n", num);
      printf("%s", str);
    }
  }while(num > 0);
}
```

Shared Memory



Shared Memory

- Kernel is not involved in data transfer
  - No need to copy data to/from the kernel
    - Very fast IPC
  - Pipes, in contrast, need to
    - Send: copy from user to kernel
    - Recv: copy from kernel to user
  - BUT, you have to *synchronize*
    - Will discuss in the next week

POSIX Shared Memory

- Sharing between unrelated processes
- APIs
  - shm_open()
    - Open or create a shared memory object
  - ftruncate()
    - Set the size of a shared memory object
  - mmap()
    - Map the shared memory object into the caller's address space

Example: Producer

```
$ ./writer /shm-name "Hello"


int main(int argc, char *argv[])
{
  char *addr;
  int fd;
  size_t len;

  fd = shm_open(argv[1], O_CREAT | O_RDWR, S_IRWXU | S_IRWXG);
  len = strlen(argv[2])+1;
  ftruncate(fd, len);
  addr = mmap(NULL, len, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
  close(fd);

  memcpy(addr, argv[2], len);
  return 0;
}
```

Example: Consumer

```
$ ./reader /shm-name


int main(int argc, char *argv[])
{
  char *addr;
  int fd;
  struct stat sb;

  fd = shm_open(argv[1], O_RDWR, 0);
  fstat(fd, &sb);
  addr = mmap(NULL, sb.st_size, PROT_READ, MAP_SHARED, fd, 0);
  close(fd);

  printf("%s\n", addr);
  return 0;
}
```

Sockets

- Sockets
  - two-way communication pipe
  - Backbone of your internet services
- Unix Domain Sockets
  - communication between processes on the same Unix system
  - special file in the file system
- Client/Server
  - client sending requests for information, processing
  - server waiting for user requests
- Socket communication modes
  - connection-based, TCP
  - connection-less, UDP

Example: Server

```
int main(int argc, char *argv[])
{
    int listenfd = 0, connfd = 0;
    struct sockaddr_in serv_addr;
    char sendBuff[1025];
    time_t ticks;

    listenfd = socket(AF_INET, SOCK_STREAM, 0);
    memset(&serv_addr, '0', sizeof(serv_addr));
    memset(sendBuff, '0', sizeof(sendBuff));

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    serv_addr.sin_port = htons(5000);

    bind(listenfd, (struct sockaddr*)&serv_addr, sizeof(serv_addr));
    listen(listenfd, 10);

    while(1)
    {
        connfd = accept(listenfd, (struct sockaddr*)NULL, NULL);
        snprintf(sendBuff, "Hello. I'm your server.");
        write(connfd, sendBuff, strlen(sendBuff));
        close(connfd);
     }
}
```

Example: Client

```
int main(int argc, char *argv[])
{
    int sockfd = 0, n = 0;
    char recvBuff[1024];
    struct sockaddr_in serv_addr;

    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    memset(&serv_addr, '0', sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(5000);

    inet_pton(AF_INET, argv[1], &serv_addr.sin_addr);
    connect(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr));

    while ( (n = read(sockfd, recvBuff, sizeof(recvBuff)-1)) > 0)
    {
        recvBuff[n] = 0;
        printf("%s\n" recvBuff);
    }
    return 0;
}
```

$ ./client 127.0.0.1

Hello. I’m your server.

Quiz

- A process produces 100MB data in memory. You want to share the data with two other processes so that each of which can access half the data (50MB each). What IPC mechanism will you use and why?

Project 1: Quash

- Goals
  - Learn to use UNIX system calls
  - Learn the concept of processes
- Write your own shell (like *bash* in Linux)
  - Run external programs
    - Use fork()/execve() system calls
  - Support built-in commands
  - Support pipe and redirections

Thread

Disclaimer: some slides are adopted from the book authors' slides with permission

Recap

- IPC
  - Shared memory
    - share a memory region between processes
    - read or write to the shared memory region

    ++ fast communication

    -- synchronization is very difficult

  - Message passing
    - exchange messages (send and receive)
    - typically involves data copies (to/from buffer)

    ++ synchronization is easier

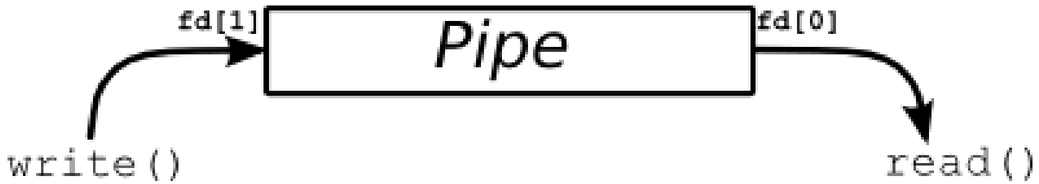
    -- slower communication

Recap

- Process
  - **Address space**
    - The process's view of memory
    - Includes program code, global variables, dynamic memory, stack
  - **Processor state**
    - Program counter (PC), stack pointer, and other CPU registers
  - **OS resources**
    - Various OS resources that the process uses
    - E.g.) open files, sockets, accounting information

Recap: Pipes

```
main()
{
  char *s, buf[1024];
  int fds[2];
  s  = "Hello World\n";

  /* create a pipe */
  pipe(fds);

  /* create a new process using fork */
  if (fork() == 0) {

    /* child process. All file descriptors, including
       pipe are inherited, and copied.*/
    write(fds[1], s, strlen(s));
    exit(0);
  }

  /* parent process */
  read(fds[0], buf, strlen(s));
  write(1, buf, strlen(s));
}
```



(*) Img. source: http://beej.us/guide/bgipc/output/html/multipage/pipes.html

Concurrent Programs

- Objects (tanks, planes, …) are moving simultaneously
- Now, imagine you implement each object as a process. Any problems?

Why Processes Are Not Always Ideal?

- Not memory efficient
  - Own address space (page tables)
  - OS resources: open files, sockets, pipes, …
- Sharing data between processes is not easy
  - No direct access to others' address space
  - Need to use IPC mechanisms

Better Solutions?

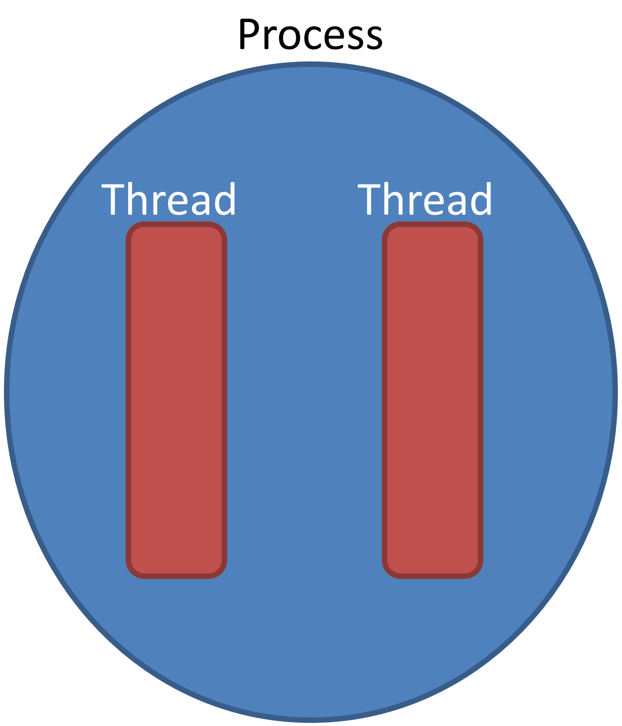
- We want to run things concurrently
  - i.e., multiple independent flows of control
- We want to share memory easily
  - Protection is not really big concern
  - Share code, data, files, sockets, …
- We want do these things efficiently
  - Don’t want to waste memory
  - Performance is very important

Thread

Thread in OS

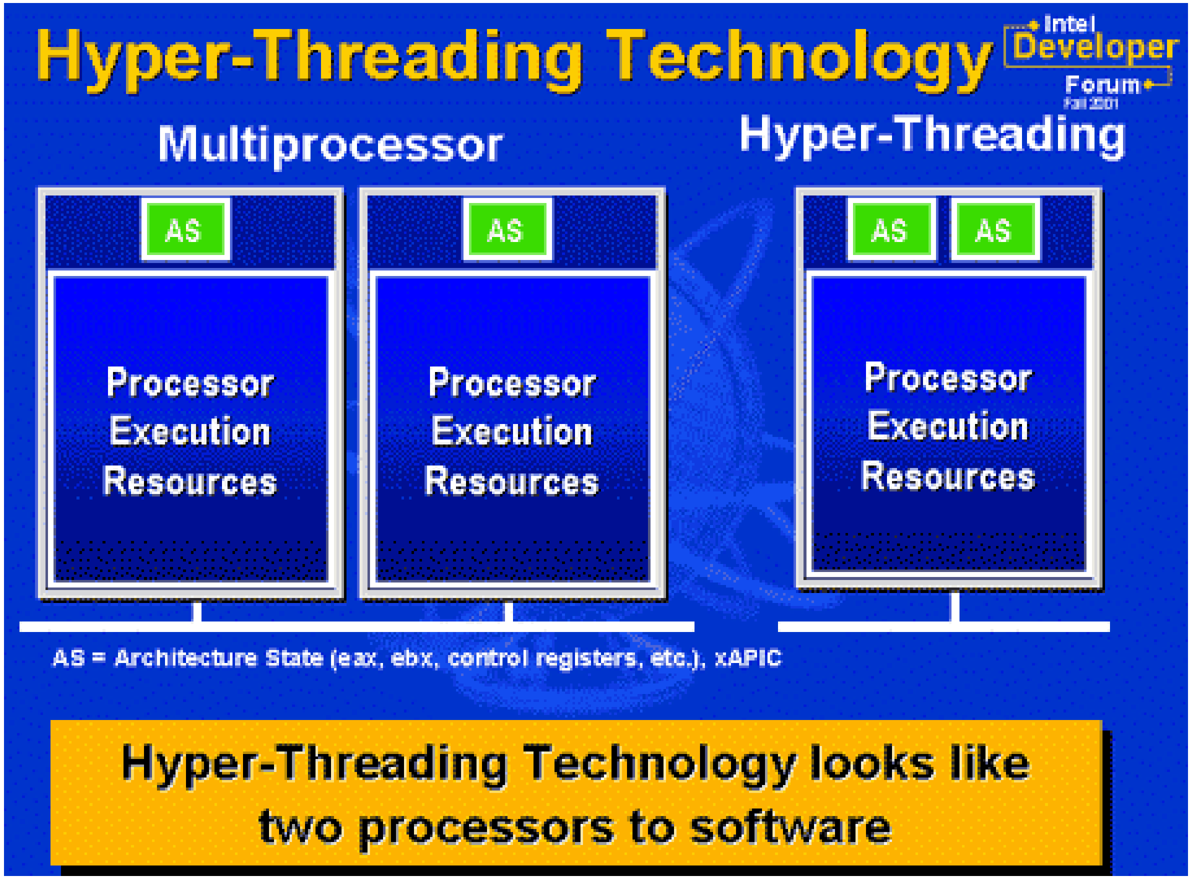
- Lightweight process

- Process
  - Address space
  - CPU context: PC, registers, stack, …
  - OS resources
- Thread
  - ~~Address space~~
  - CPU context: PC, registers, stack, …
  - ~~OS resources~~



Thread in Architecture

- Logical processor



http://www.pcstats.com/articleview.cfm?articleID=1302

Thread

- Lightweight process
  - Own independent flow of control (execution)
  - Stack, thread specific data (tid, …)
  - Everything else (address space, open files, …) is shared

Shared	Private
- Program code - (Most) data - Open files, sockets, pipes - Environment (e.g., HOME)	- Registers - Stack - Thread specific data - Return value

KU THE UNIVERSITY OF KANSAS

Process vs. Thread

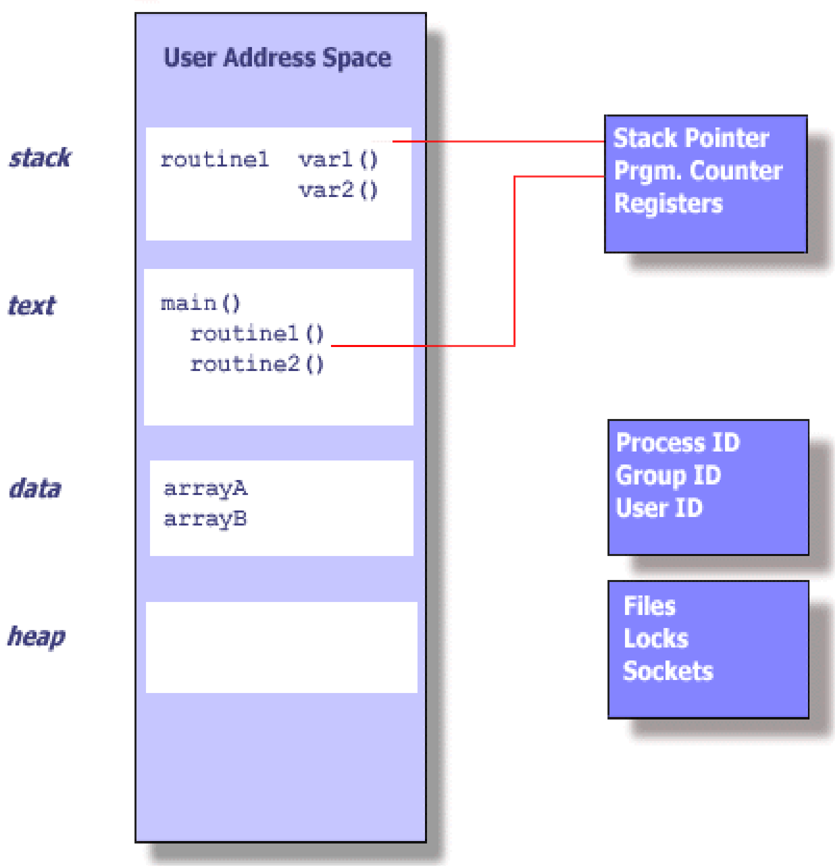


Figure source: https://computing.llnl.gov/tutorials/pthreads/

Process vs. Thread

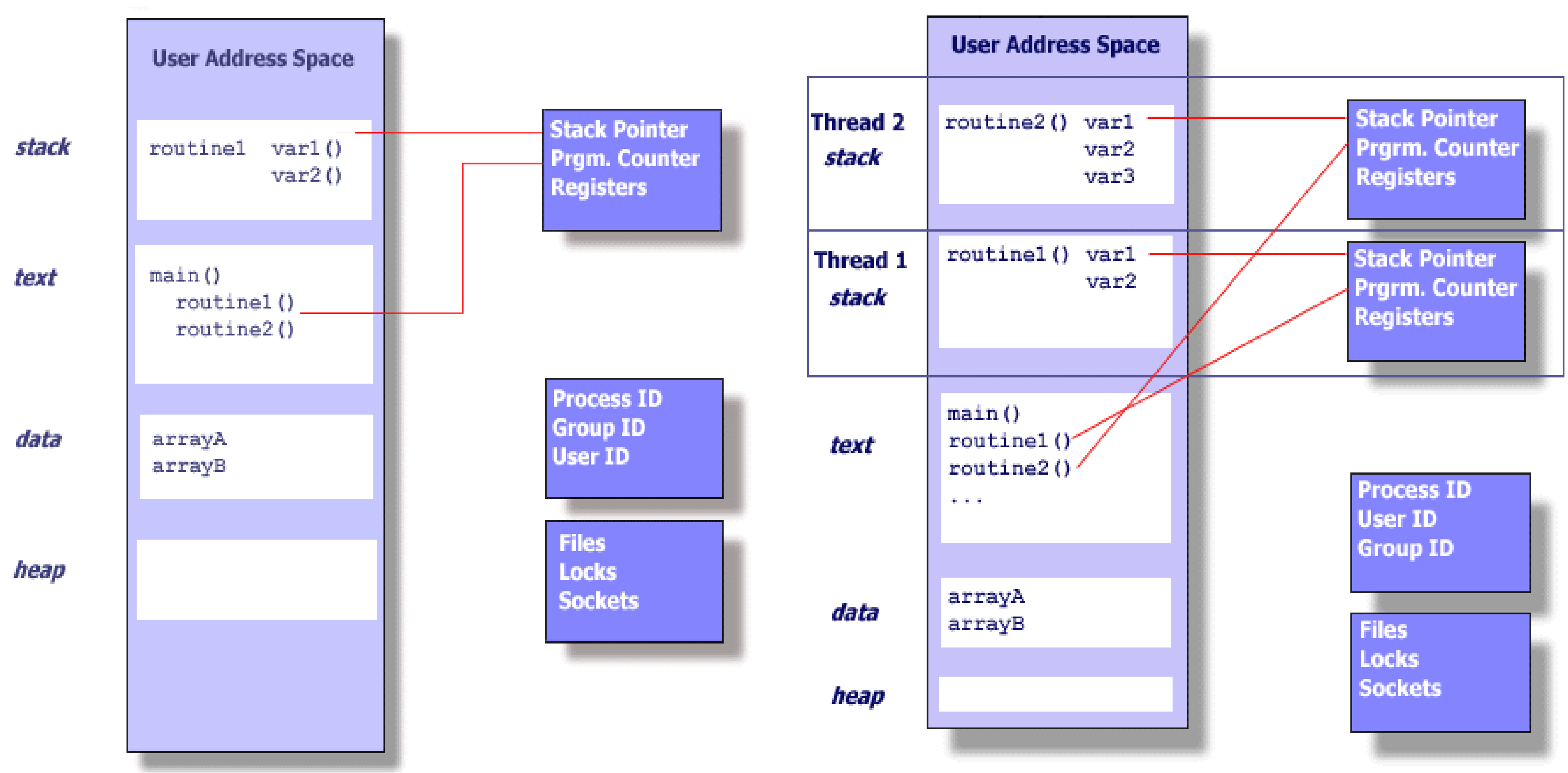


Figure source: https://computing.llnl.gov/tutorials/pthreads/

Thread Benefits

- Responsiveness
  - Simple model for concurrent activities.
  - No need to block on I/O
- Resource Sharing
  - Easier and faster memory sharing (but be aware of synchronization issues)
- Economy
  - Reduces context-switching and space 降低开销 overhead → better performance
- Scalability
  - Exploit multicore CPU

Thread Programming in UNIX

- Pthread
  - IEEE POSIX standard threading API
- Pthread API
  - Thread management
    - create, destroy, detach, join, set/query thread attributes
  - Synchronization
    - Mutexes –lock, unlock
    - Condition variables – signal/wait

Pthread API

- pthread_attr_init – initialize the thread attributes object
  - int pthread_attr_init(pthread_attr_t *attr);
  - defines the attributes of the thread created
- pthread_create – create a new thread
  - int pthread_create(pthread_t *restrict thread, const pthread_attr_t *restrict attr, void *(*start_routine)(void*), void *restrict arg);
  - upon success, a new thread id is returned in thread
- pthread_join – wait for thread to exit
  - int pthread_join(pthread_t thread, void **value_ptr);
  - calling process blocks until thread exits
- pthread_exit – terminate the calling thread
  - void pthread_exit(void *value_ptr);
  - make return value available to the joining thread

Pthread Example 1

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

int sum; /* data shared by all threads */
void *runner (void  *param)
{
    int i, upper = atoi(param);
    sum = 0;
    for(i=1 ; i<=upper ; i++)
        sum += i;
    pthread_exit(0);
}

int main (int argc, char *argv[])
{
    pthread_t tid; /* thread identifier */
    pthread_attr_t attr;
    pthread_attr_init(&attr);

    /* create the thread */
    pthread_create(&tid, &attr, runner, argv[1]);
    /* wait for the thread to exit */
    pthread_join(tid, NULL);
    fprintf(stdout, "sum = %d\n", sum);
}
```

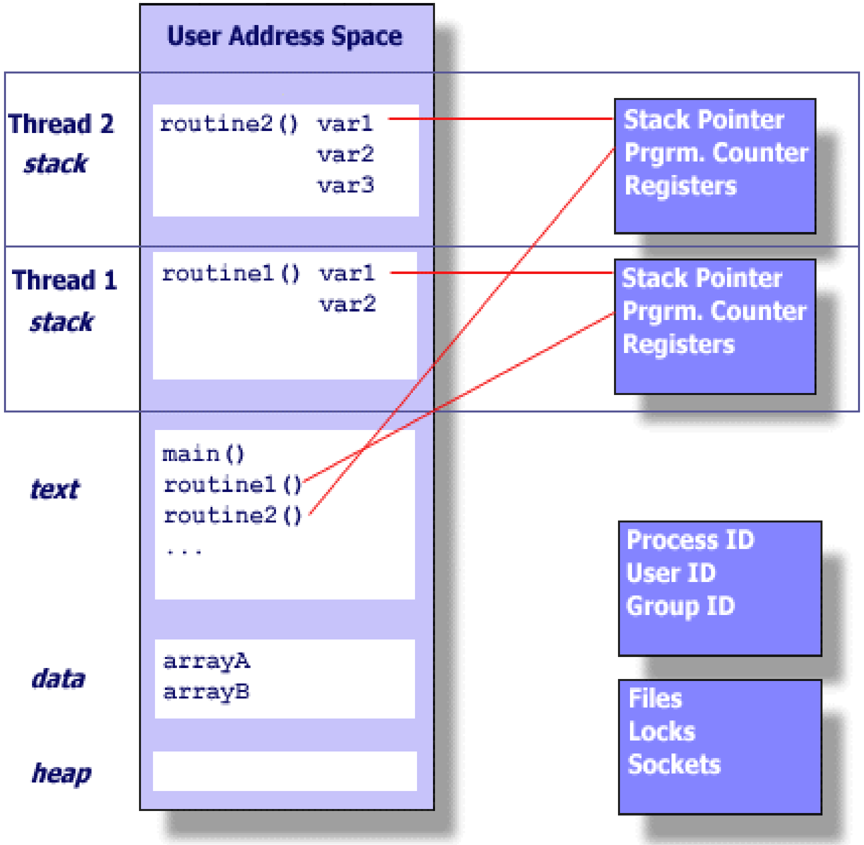
Quiz: Final ouput?

$./a.out 10

sum = 55

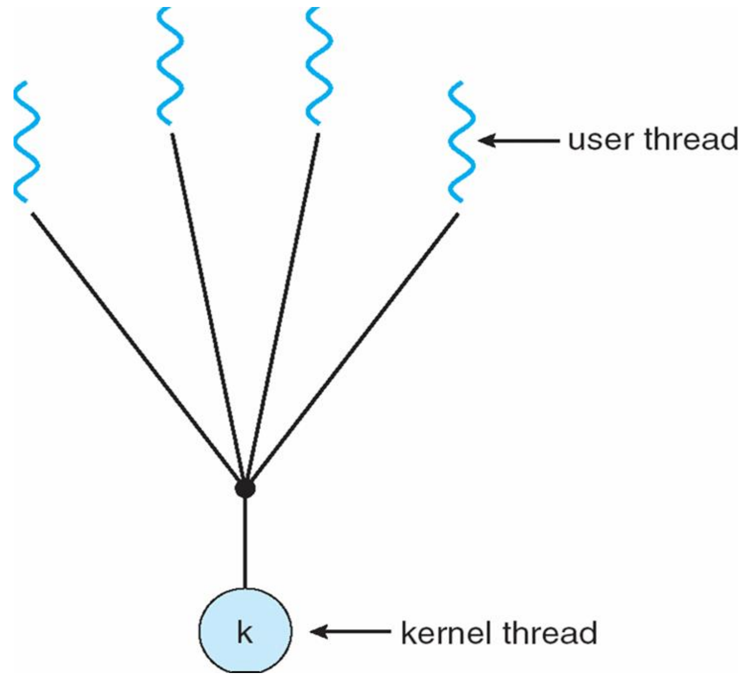
Pthread Example 2

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

int arrayA[10], arrayB[10];

void *routine1(void  *param)
{
    int var1, var2
    …
}
void *routine2(void  *param)
{
    int var1, var2, var3
    …
}


int main (int argc, char *argv[])
{
    /* create the thread */
    pthread_create(&tid[0], &attr, routine1, NULL);
    pthread_create(&tid[1], &attr, routine2, NULL);
    pthread_join(tid[0]); pthread_join(tid[1]);
}
```



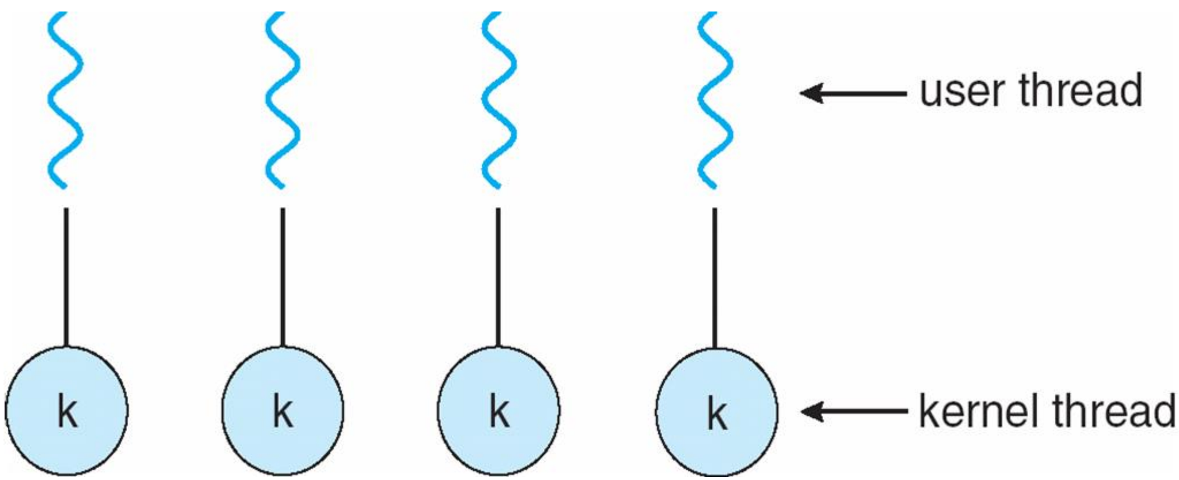
User-level Threads

- Kernel is unaware of threads
  - Early UNIX and Linux did not support threads
- Threading runtime
  - Handle context switching
    - Setjmp/longjmp, …
- Advantage
  - No kernel support
  - Fast (no kernel crossing)
- Disadvantage
  - Blocking system call. What happens?



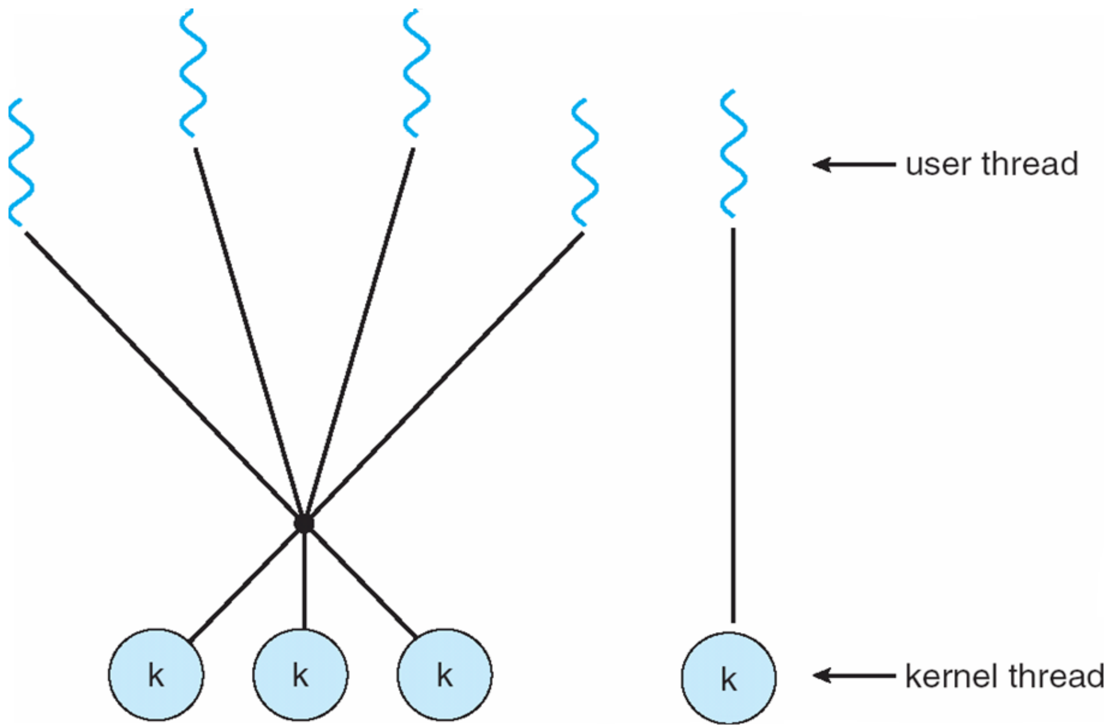
Kernel-level Threads

- Native kernel support for threads
  - Most modern OS (Linux, Windows NT)
- Advantage
  - No threading runtime
  - Native system call handing
- Disadvantage
  - Overhead



Hybrid Threads

- Many kernel threads to many user threads
  - Best of both worlds?



Threads: Advanced Topics

- Semantics of Fork/exec()
- Signal handling
- Thread pool
- Multicore

Semantics of fork()/exec()

- Remember fork(), exec() system calls?
  - Fork: create a child process (a copy of the parent)
  - Exec: replace the address space with a new pgm.

- Duplicate *all* threads or *the caller* only?
  - Linux: the calling thread only
  - Complicated. Don't do it!
    - Why? Mutex states, library, …
    - Exec() immediately after Fork() may be okay.

Signal Handling

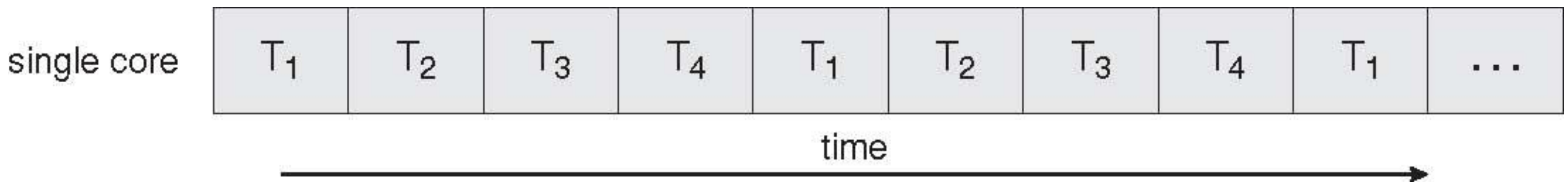
- What is *Singal*?
  - $ man 7 signal
  - OS to process notification
    - "hey, wake-up, you've got a packet on your socket,"
    - "hey, wake-up, your timer is just expired."
- Which *thread* to deliver a signal?
  - Any thread
    - e.g., kill(pid)
  - Specific thread
    - E.g., pthread_kill(tid)

Thread Pool

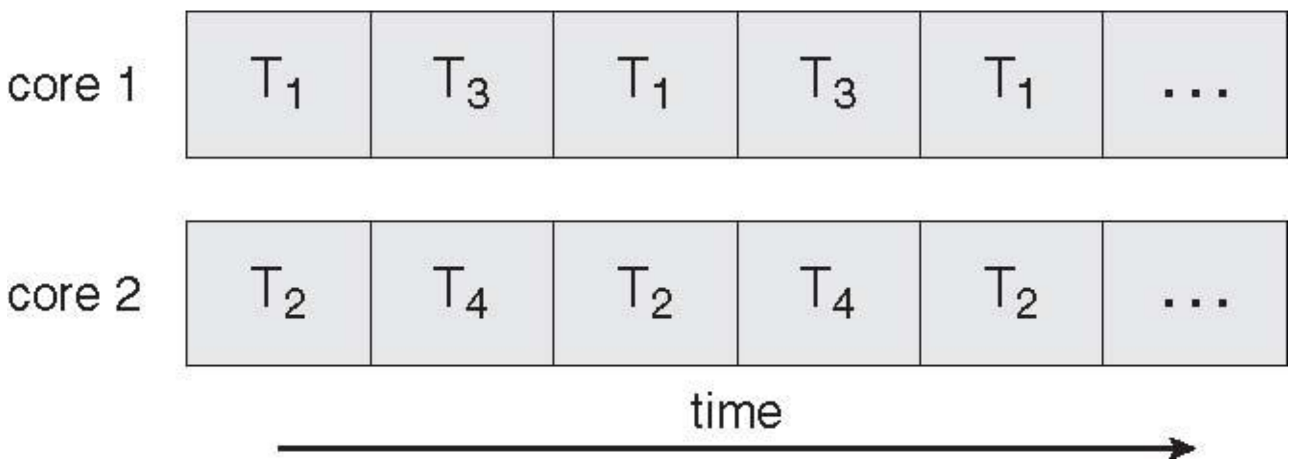
- Managing threads yourself can be cumbersome and costly
  - Repeat: create/destroy threads as needed.

- Let's create a set of threads ahead of time, and just ask them to execute my functions
  - #of thread ~ #of cores
  - No need to create/destroy many times
  - Many high-level parallel libraries use this.
    - e.g., Intel TBB (threading building block), …

Single Core Vs. Multicore Execution



Single core execution



Multiple core execution

Challenges for Multithreaded Programming in Multicore

- How to divide activities?
- How to divide data?
- **How to synchronize accesses to the shared data? → next class**
- **How to test and dubug?**

EECS750

Summary

- Thread
  - What is it?
    - Independent flow of control.
  - What for?
    - Lightweight programming construct for concurrent activities
  - How to implement?
    - Kernel thread vs. user thread
- Next class
  - How to synchronize?