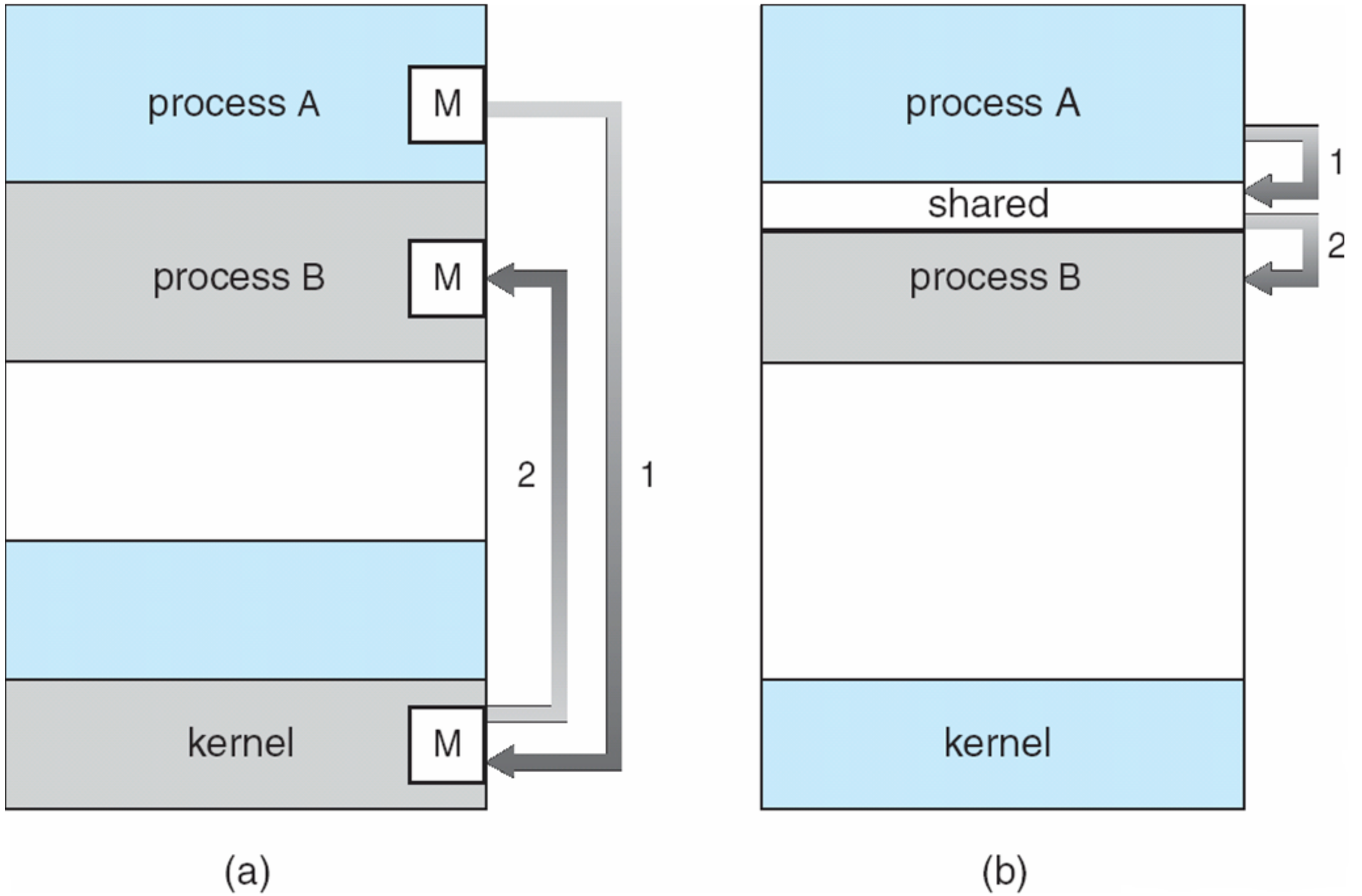
Processes

Inter-process communication

Cooperating Processes

- **Independent processes**: execute independently of other processes.
- **Cooperating processes**: depend on the execution of other processes.
  - Cooperation requires inter-process communication.

Models of IPC



Posix Shared Memory

1. Process first creates shared memory segment

```
segment id = shmget(IPC PRIVATE, size, S_IRUSR |
S_IWUSR);
```

2. Process wanting access to the shared memory must attach to it

```
shared_memory = (char *) shmat(id, NULL, 0);
```

3. Now the process could write to the shared memory

```
sprintf(shared_memory, "Writing to shared memory");
```

4. When done, a process can detach the shared memory from its address space

```
shmdt(shared_memory);
```

shm_server.c

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>

#define SHMSZ     27

main()
{
    char c;
    int shmid;
    key_t key;
    char *shm, *s;

    /*
     * We'll name our shared memory segment
     * "5678".
     */
    key = 5678;

    /*
     * Create the segment.
     */
    if ((shmid = shmget(key, SHMSZ,
                IPC_CREAT | 0666)) < 0)
    {
        perror("shmget");
        exit(1);
    }

    /*
     * attach the segment to our data space.
     */
    if ((shm = shmat(shmid, NULL, 0)) ==
        (char *) -1)
    {
        perror("shmat");
        exit(1);
    }

    /* put things into the memory for the
     * other process to read.*/
    s = shm;

    for (c = 'a'; c <= 'z'; c++)
        *s++ = c;
    *s = NULL;

    /* Finally, wait until the other process
     * changes the first character of our
     * memory to '*', indicating that it
     * has read what we put there. */
    while (*shm != '*')
        sleep(1);

    exit(0);
}
```

shm_client.c

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>

#define SHMSZ     27

main()
{
    int shmid;
    key_t key;
    char *shm, *s;

    /*
     * We need to get the segment named
     * "5678", created by the server.
     */
    key = 5678;

    /*
     * Locate the segment.
     */
    if((shmid=shmget(key,SHMSZ,0666)) < 0)
    {
        perror("shmget");
        exit(1);
    }
```

```
    /*
     * attach the segment to our data space.
     */
    if ((shm = shmat(shmid, NULL, 0)) ==
         (char *) -1)
    {
        perror("shmat");
        exit(1);
    }

    /*
     * read what the server put in
     * the memory.
     */
    for (s = shm; *s != NULL; s++)
        putchar(*s);
    putchar('\n');

    /*
     * Finally, change the first character
     * of the segment to '*', indicating
     * we have read the segment.
     */
    *shm = '*';

    exit(0);
}
```

Message Passing

Show programs:

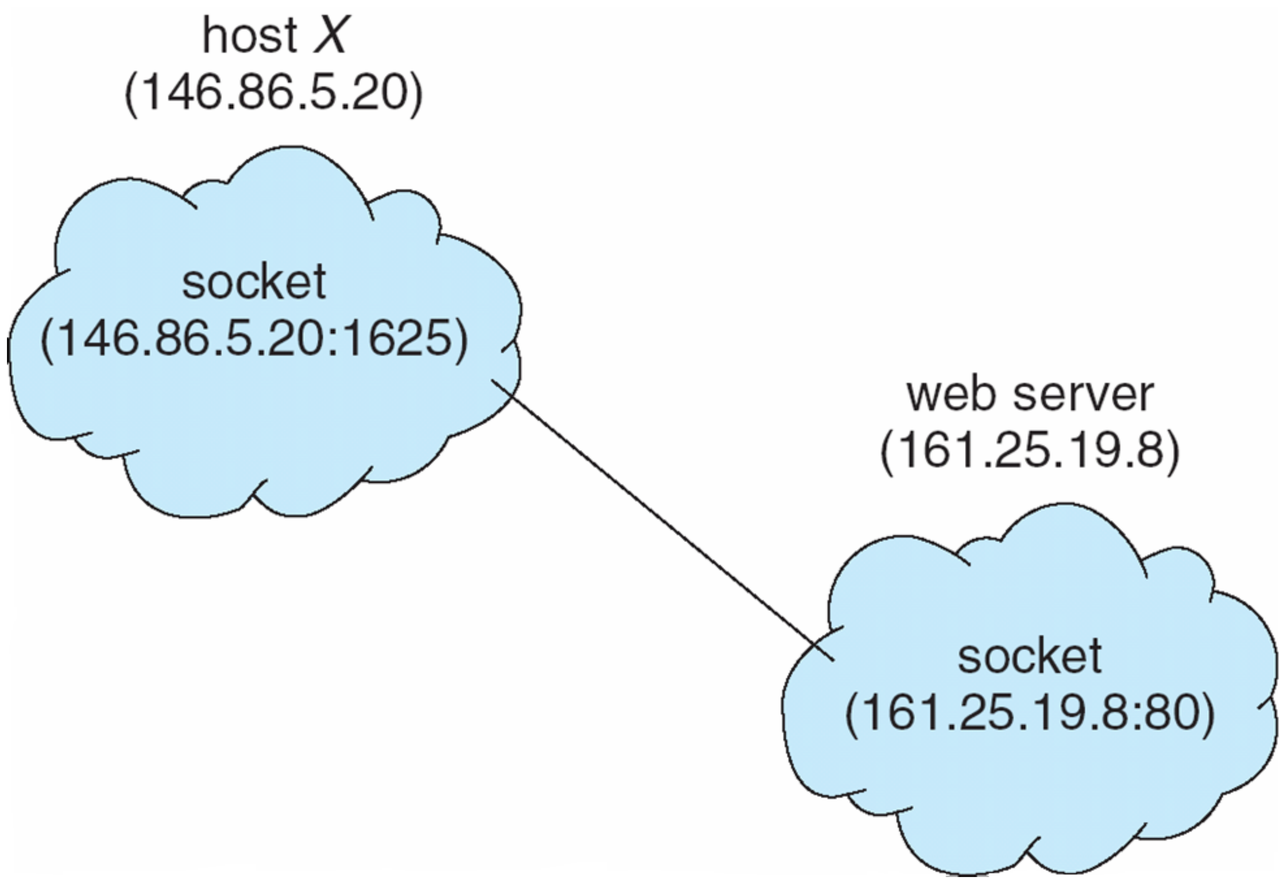
- `message_send.c`
- `message_rec.c`

Processes

Sockets (communicating using the network)

Sockets

- A **socket** is defined as an *endpoint for communication*
- Concatenation of **IP address** and **port**
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- Communication consists between a pair of sockets



Sockets

- Most interprocess communication uses the client-server model.
- Client needs to know of the existence of and the address of the server, but the server does not need to know the address of (or even the existence of) the client prior to the connection being established.
- Once a connection is established, both sides can send and receive information.
- A socket is one end of an interprocess communication channel. The two processes each establish their own socket.

Socket address domains

- Two processes can communicate with each other only if their sockets are of the same type and in the same domain.
- There are two widely used address domains, each has its own address format:
  - the unix domain: two processes share a common file system.
  - the Internet domain: two processes running on any two hosts on the Internet.

Sockets

- Connection steps on the client side
- Connection steps on the server side (single connection)

Sockets

Connection steps on the client side:

1. Create a socket with the `socket()` system call.

2. Connect the socket to the address of the server using the `connect()` system call.

3. Send and receive data. There are many of ways to do this. The simplest is to use the `read()` and `write()` system calls.

Sockets

Connection steps on the server side (single connection):

1. Create a socket with the `socket()` system call
2. Bind the socket to an address using the `bind()` system call. For a server socket on the Internet, an address consists of a port number on the host machine.
3. Listen for connections with the `listen()` system call
4. Accept a connection with the `accept()` system call. This call typically blocks until a client connects with the server.
5. Send and receive data

Socket types

Two widely used socket types:

- **stream sockets**: communicate via a continuous stream of characters. Stream sockets use TCP (Transmission Control Protocol), which is a reliable, stream-oriented protocol.
- **datagram sockets**: read entire messages at once. Use UDP (Unix Datagram Protocol), which can be unreliable and message oriented.

in CSE4001, we will work with sockets in the Internet domain using the TCP protocol.

Examples of connecting using sockets

- client-server with single connection.
- server forking multiple "handler" processes for each client connection.

```
while (1)
  {
    newsockfd = accept(sockfd,
                (struct sockaddr *) &cli_addr, &clilen);
    if (newsockfd < 0)
      error("ERROR on accept");
    pid = fork();
    if (pid < 0)
      error("ERROR on fork");
    if (pid == 0)
    {
      close(sockfd);
      dostuff(newsockfd);
      exit(0);
    }
    else
      close(newsockfd);
  } /* end of while */
```

Server forking multiple "handler" processes for each client connection.

Accepting connection goes inside an infinite loop.

http://www.linuxhowtos.org/C_C++/socket.htm

Server forking multiple "handler" processes for each client connection.

```
while (1)
 {
   newsockfd = accept(sockfd,
               (struct sockaddr *) &cli_addr, &clilen);
   if (newsockfd < 0)
     error("ERROR on accept");
   pid = fork();
   if (pid < 0)
     error("ERROR on fork");
   if (pid == 0)
   {
     close(sockfd);
     dostuff(newsockfd);
     exit(0);
   }
   else
     close(newsockfd);
 } /* end of while */
```

Connection is established. Create new process to handle the service.

http://www.linuxhowtos.org/C_C++/socket.htm

Server forking multiple "handler" processes for each client connection.

```
while (1)
 {
   newsockfd = accept(sockfd,
               (struct sockaddr *) &cli_addr, &clilen);
   if (newsockfd < 0)
     error("ERROR on accept");
   pid = fork();
   if (pid < 0)
     error("ERROR on fork");
   if (pid == 0)
   {
     close(sockfd);
     dostuff(newsockfd);
     exit(0);
   }
   else
     close(newsockfd);
 } /* end of while */
```

Child process will close **`sockfd`** and call the handling function passing **`newsockfd`** as argument.

Once the communication between client and handler is completed, child exits.

http://www.linuxhowtos.org/C_C++/socket.htm

Server forking multiple "handler" processes for each client connection.

```
while (1)
 {
   newsockfd = accept(sockfd,
               (struct sockaddr *) &cli_addr, &clilen);
   if (newsockfd < 0)
     error("ERROR on accept");
   pid = fork();
   if (pid < 0)
     error("ERROR on fork");
   if (pid == 0)
   {
     close(sockfd);
     dostuff(newsockfd);
     exit(0);
   }
   else
     close(newsockfd);
 } /* end of while */
```

Parent closes `newsockfd` and returns to `accept()` to wait for a new connection.

http://www.linuxhowtos.org/C_C++/socket.htm

The return of the zombies

A zombie is a process that has terminated but but cannot be permitted to fully die because at some point in the future, the parent of the process might execute a **`wait()`** and would want information about the death of the child.

Figure from: https://scottlinux.com/2012/01/13/how-to-kill-zombie-processes/

http://www.linuxhowtos.org/C_C++/socket.htm

The invasion of the zombies

Problem with the previous code:

- Each of these connections will create a zombie when the connection is terminated.
- When a child dies, it sends a `SIGCHLD` signal to its parent. But, the handling of this signal is system dependent.

http://www.linuxhowtos.org/C_C++/socket.htm

SunOS: Example of catching SIGCHLD and avoiding zombies

```
void proc_exit() {
    int wstat;
    union wait wstat;
    pid_t pid;

    while (TRUE) {
      pid = wait3 (&wstat, WNOHANG, (struct rusage *)NULL );
      if (pid == 0)
        return;
      else if (pid == -1)
        return;
      else
        printf ("Return code: %d\n", wstat.w_retcode);
    }
}
int main () {
    signal (SIGCHLD, proc_exit);
    switch (fork()) {
      case -1:
        perror ("main: fork");
        exit (0);
      case 0:
        printf ("I'm alive (temporarily)\n");
        exit (rand());
      default:
        pause();
    }
}
```

http://g-ecx.images-amazon.com/images/G/01/mas/prod/images/acontent/B0067VKQLE/5-right1._SL312_V1625499986_.jpg

Credits:

Slides on socket communication based on the sockets tutorial from:

http://www.linuxhowtos.org/C_C++/socket.htm