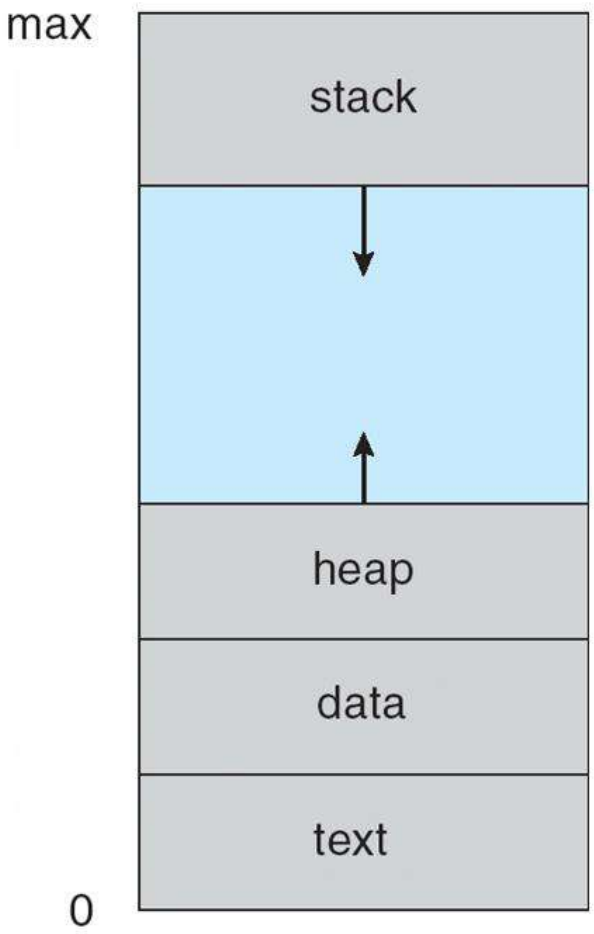
Processes

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- Examples of IPC Systems
- Communication in Client-Server Systems

Process Concept

- An operating system executes a variety of programs:
  - Batch system –jobs
  - Time-shared systems –user programs or tasks
- Textbook uses the terms *job and process almost interchangeably*
- Process –a program in execution; process execution must progress in sequential fashion
- A process includes:
  - program counter
  - Stack
  - data section

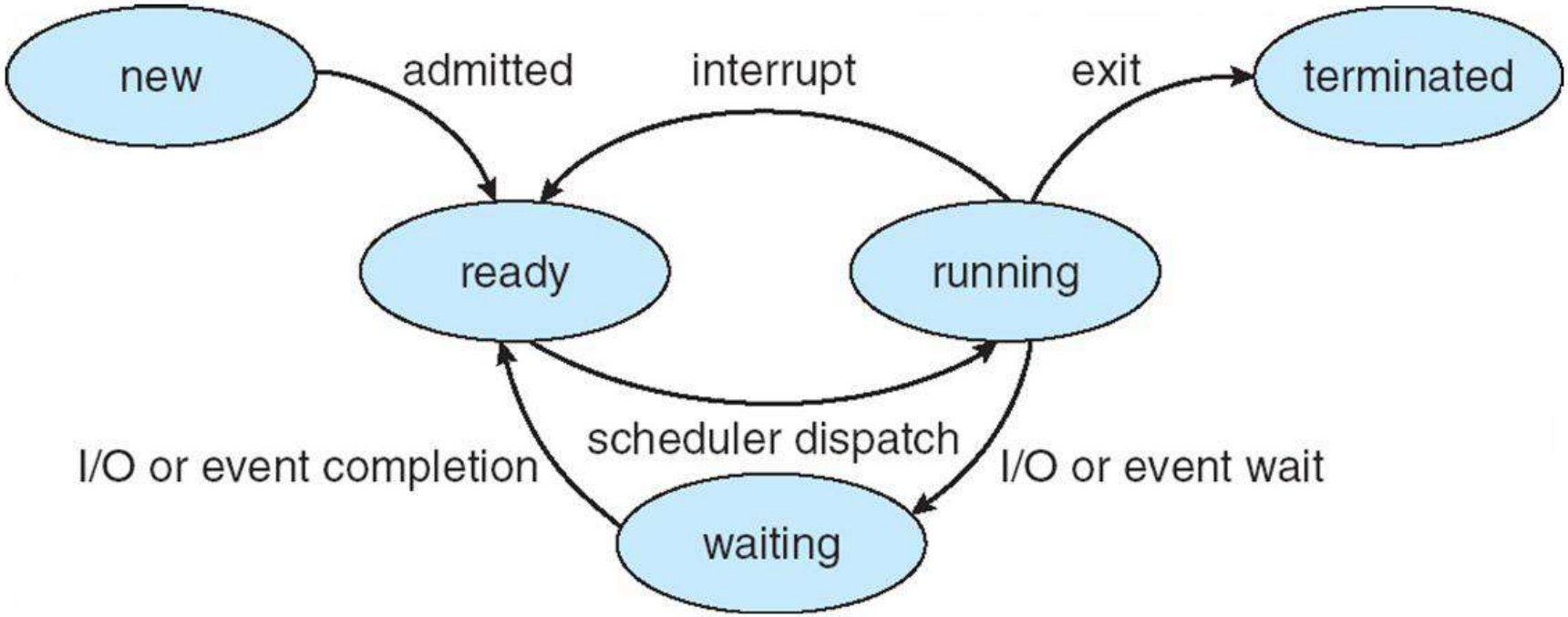
Process in Memory



Process State

- As a process executes, it changes *state*
  - **new**: The process is being created
  - **running:** Instructions are being executed
  - **waiting:** The process is waiting for some event to occur
  - **ready**: The process is waiting to be assigned to a processor
  - **terminated:** The process has finished execution

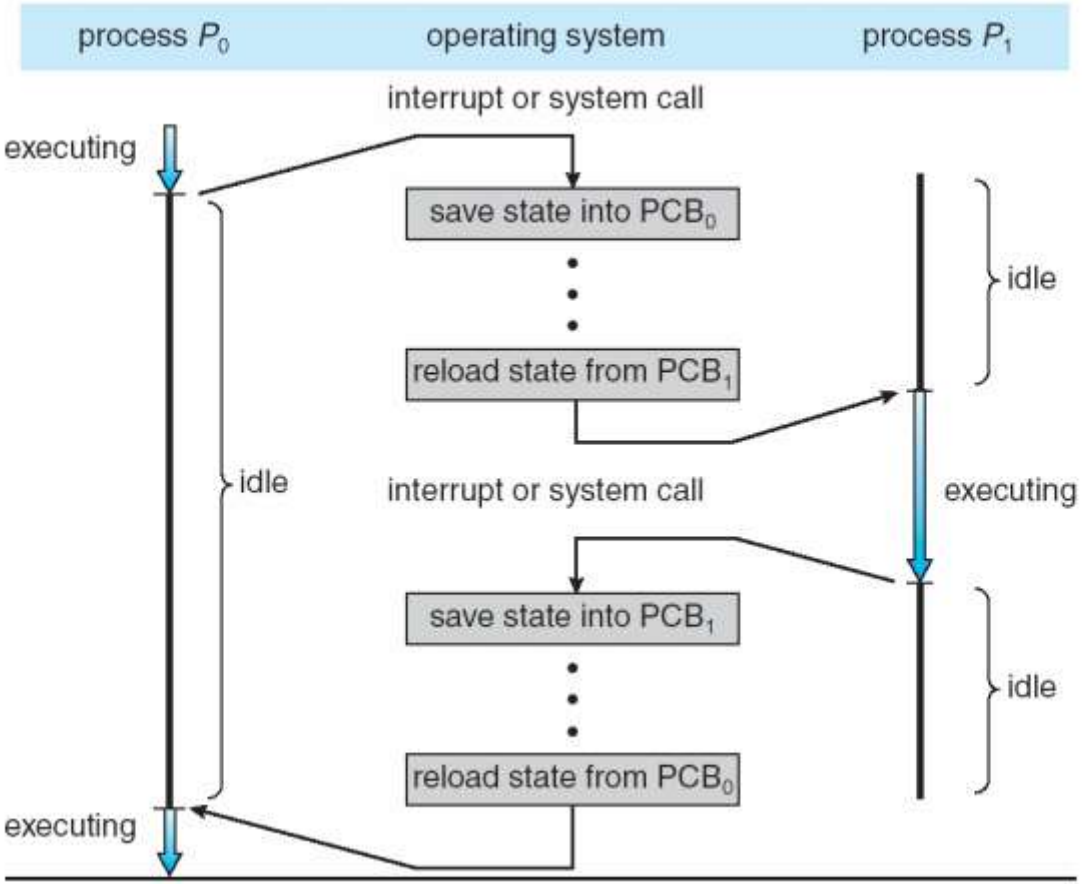
Process State



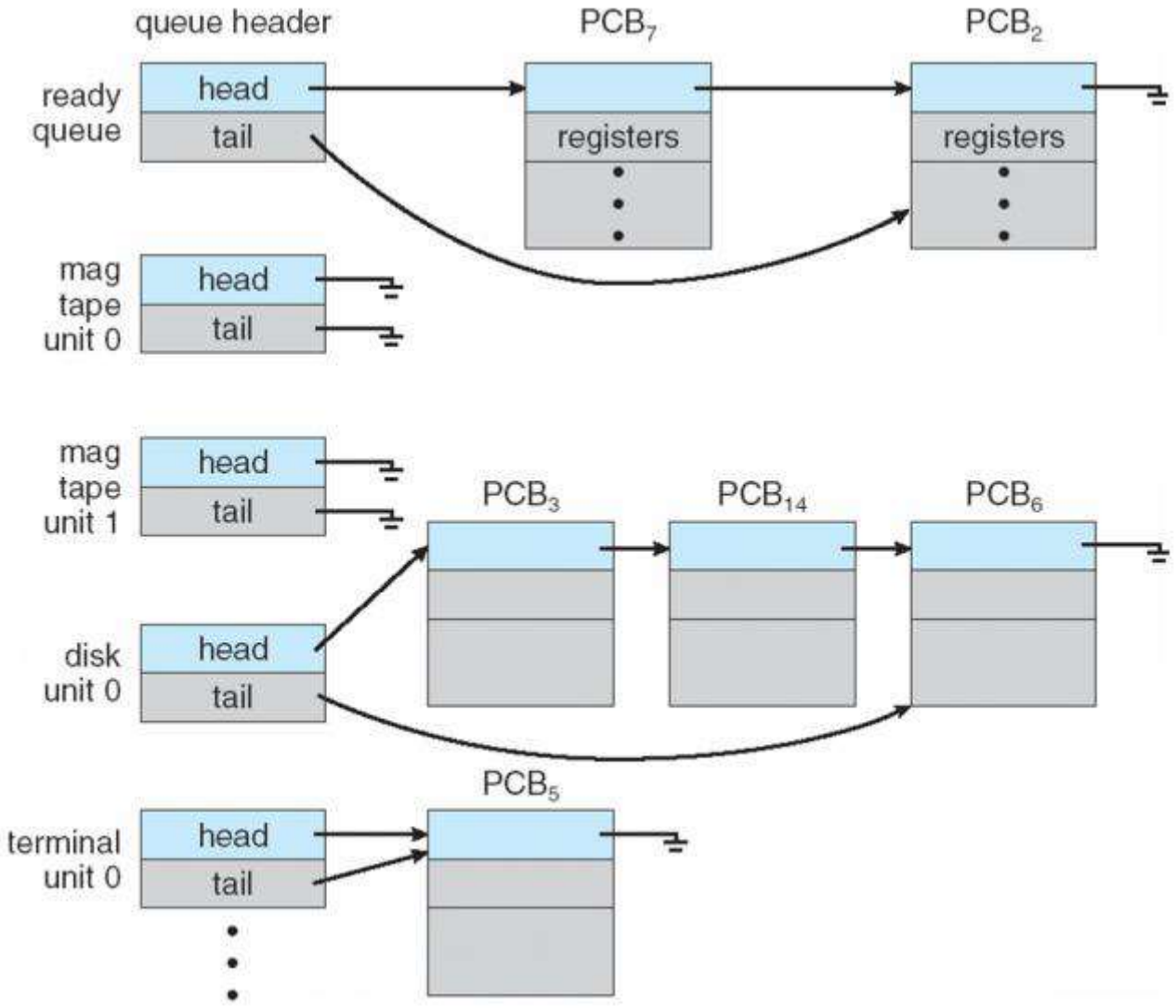
Process Control Block (PCB)

- Information associated with each process
  - Process state
  - Program counter
  - CPU registers
  - CPU scheduling information
  - Memory-management information
  - I/O status information

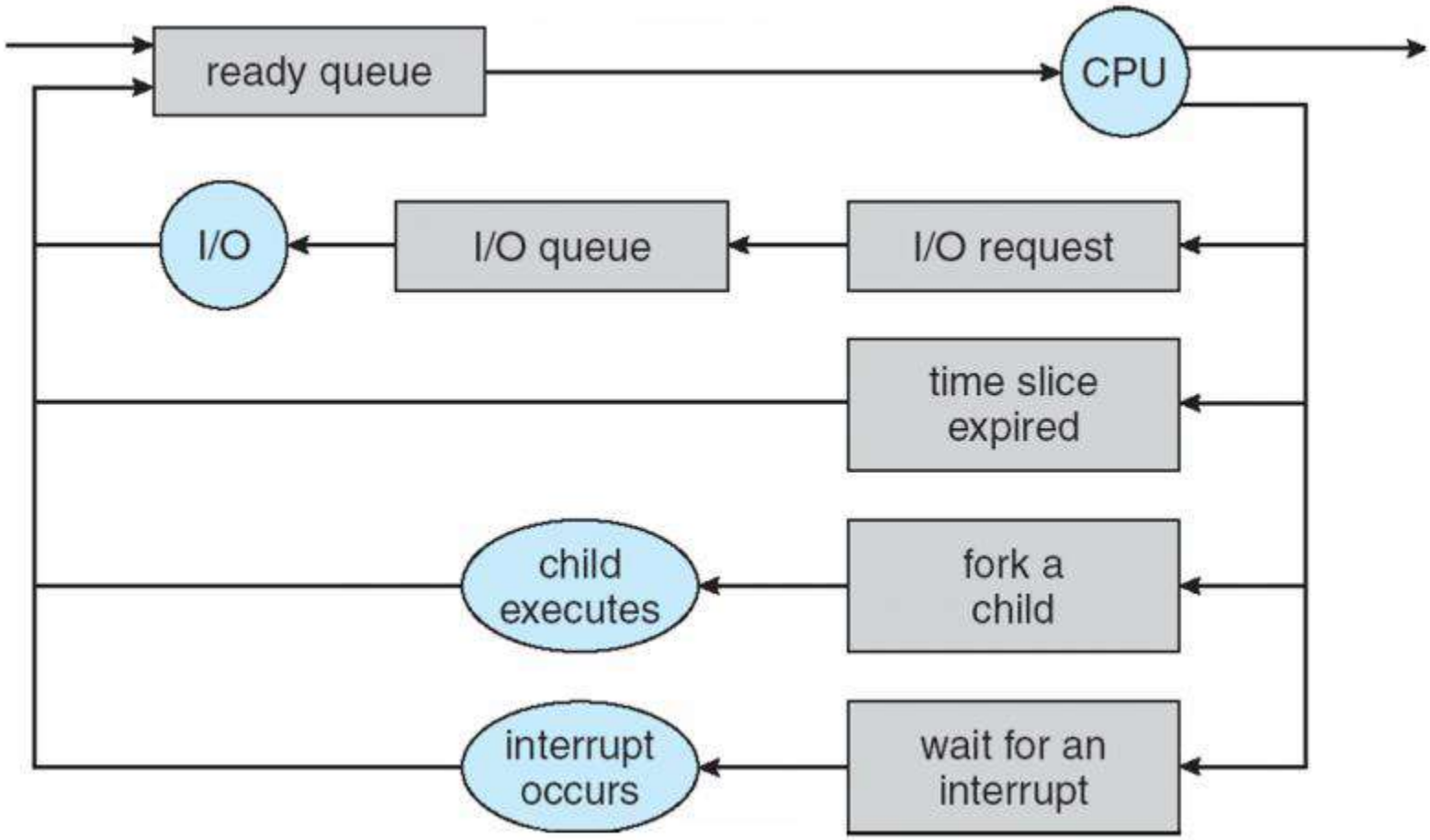
Process Control Block (PCB)


process P_0
operating system
process P_1
interrupt or system call
executing
save state into PCB_0
reload state from PCB_1
idle
idle
interrupt or system call
executing
save state into PCB_1
reload state from PCB_0
idle
executing

Ready Queue And Various I/O Queues



Representation of Process Scheduling



Schedulers

- **Long-term scheduler(or job scheduler) – selects which processes should be brought into the ready queue**
- **Short-term scheduler(or CPU scheduler) – selects which process should be executed next and allocates CPU**

Schedulers

- Short-term scheduler is invoked very frequently (milliseconds) (must be fast)
- Long-term scheduler is invoked very infrequently (seconds, minutes) (may be slow)
- The long-term scheduler controls the *degree of multiprogramming*
- Processes can be described as either:
  - **I/O-bound process–spends more time doing I/O than computations, many short CPU bursts**
  - **CPU-bound process–spends more time doing computations; few very long CPU bursts**

Context Switch

- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a context switch
- Context of a process represented in the PCB
- Context-switch time is overhead; the system does no useful work while switching
- Time dependent on hardware support

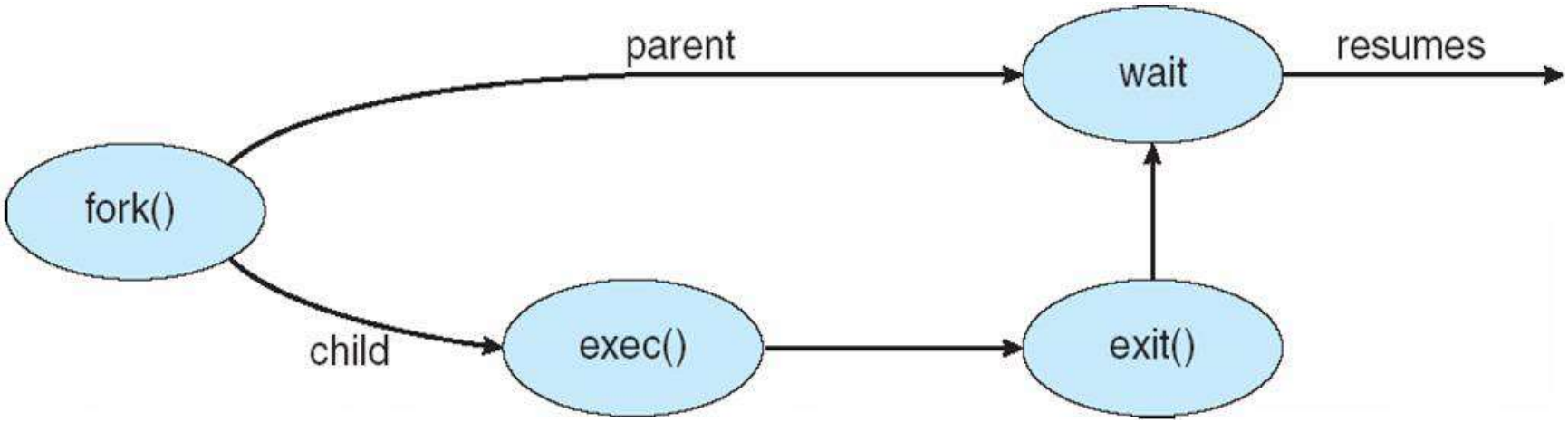
Process Creation

- **Parent process create children processes, which, in turn create other processes, forming a tree of processes**
- Generally, process identified and managed via **a process identifier (pid)**
- Resource sharing
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution
  - Parent and children execute concurrently
  - Parent waits until children terminate

Process Creation

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - **Fork** system call creates new process
  - **Exec** system call used after a fork to replace the process' memory space with a new program

Process Creation



C program - fork

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

int main()
{
  int i, pid;

  printf("My process ID is %d\n", getpid());

  for ( i = 0; i < 3; i++ ) {
    pid = fork();

    if ( pid != 0 ) {
      printf("I am the parent => PID = %d, child ID = %d\n", getpid(), pid);
}
    else {
      printf("I am the child  => PID = %d\n", getpid());

    }
  }
  while(1);

  return(0);
}
```

C program - fork

```
#include <stdio.h>

int main()
{
  int i, pid;

  printf("My process ID is %d\n", getpid());

  for ( i = 0; i < 3; i++ ) {
    pid = fork();

    if ( pid == 0 ) {
      printf("I am the child  => PID = %d\n", getpid());
      while(1);
    }
    else
      printf("I am the parent => PID = %d, child ID = %d\n", getpid(), pid);
  }
  while(1);
  return(0);
}
```

C program - fork

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int main()
{
  int i, status;
  pid_t pid, pid_array[3];

  printf("My process ID is %d\n", getpid());
  for ( i = 0; i < 3; i++ ) {
    pid = fork();
    if ( pid == 0 ) {
      printf("I am the child  => PID = %d\n", getpid());
      while(1);
    }
    else {
      printf("I am the parent => PID = %d, child ID = %d\n", getpid(), pid);
      pid_array[i] = pid;
    }
  }
  //while( 1 ) {
    for ( i = 0; i < 3; i++ ) {
      if (pid = wait(&status)) {
         printf("Process ID %d has terminated\n", pid);
      }
    }
```

C program - fork

```
int main()
{
pid_t pid;
/* fork another process */
pid = fork();
if (pid < 0) { /* error occurred */
fprintf(stderr, "Fork Failed");
exit(-1);
}
else if (pid == 0) { /* child process */
execlp("/bin/ls", "ls", NULL);
}
else { /* parent process */
/* parent will wait for the child to complete */
wait (NULL);
printf ("Child Complete");
exit(0);
}
}
```

Process Creation –win32

```
BOOL CreateProcess(
   LPCTSTR               lpApplicationName,      // name of executable module
   LPTSTR                lpCommandLine,          // command line string
   LPSECURITY_ATTRIBUTES lpProcessAttributes,    // SD
   LPSECURITY_ATTRIBUTES lpThreadAttributes,     // SD
   BOOL                  fInheritHandles,        // handle inheritance option
   DWORD                 dwCreationFlags,        // creation flags
   LPVOID                lpEnvironment,          // new envirnment block
   LPCTSTR               lpCurrentDirectory,     // current directory name
   LPSTARTUPINFO         lpStartupInfo,          // startup information
   LPPROCESS_INFORMATION lpProcessInformation    // process
     information
);
```

Process Creation –win32

```
STARTUPINFO         si ;
PROCESS_INFORMATION  pi;
    ZeroMemory(&si,sizeof(si));
    si.cb=sizeof(si);
    ZeroMemory(&si,sizeof(pi));

    char              szExe[MAX_PATH] = “mspaint.exe";
 ///// HINSTANCE hInstance = GetModuleHandle(NULL);
////  GetModuleFileName(hInstance, szExe, MAX_PATH);
    if(!CreateProcess(0, szExe, 0, 0, FALSE, 0, 0, 0, &si, &pi))
          {    printf(“Create Process failed”); return -1;
          }
     // optionally wait for process to finish
     WaitForSingleObject(pi.hProcess, INFINITE);
    printf(“Child complete”);
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
```

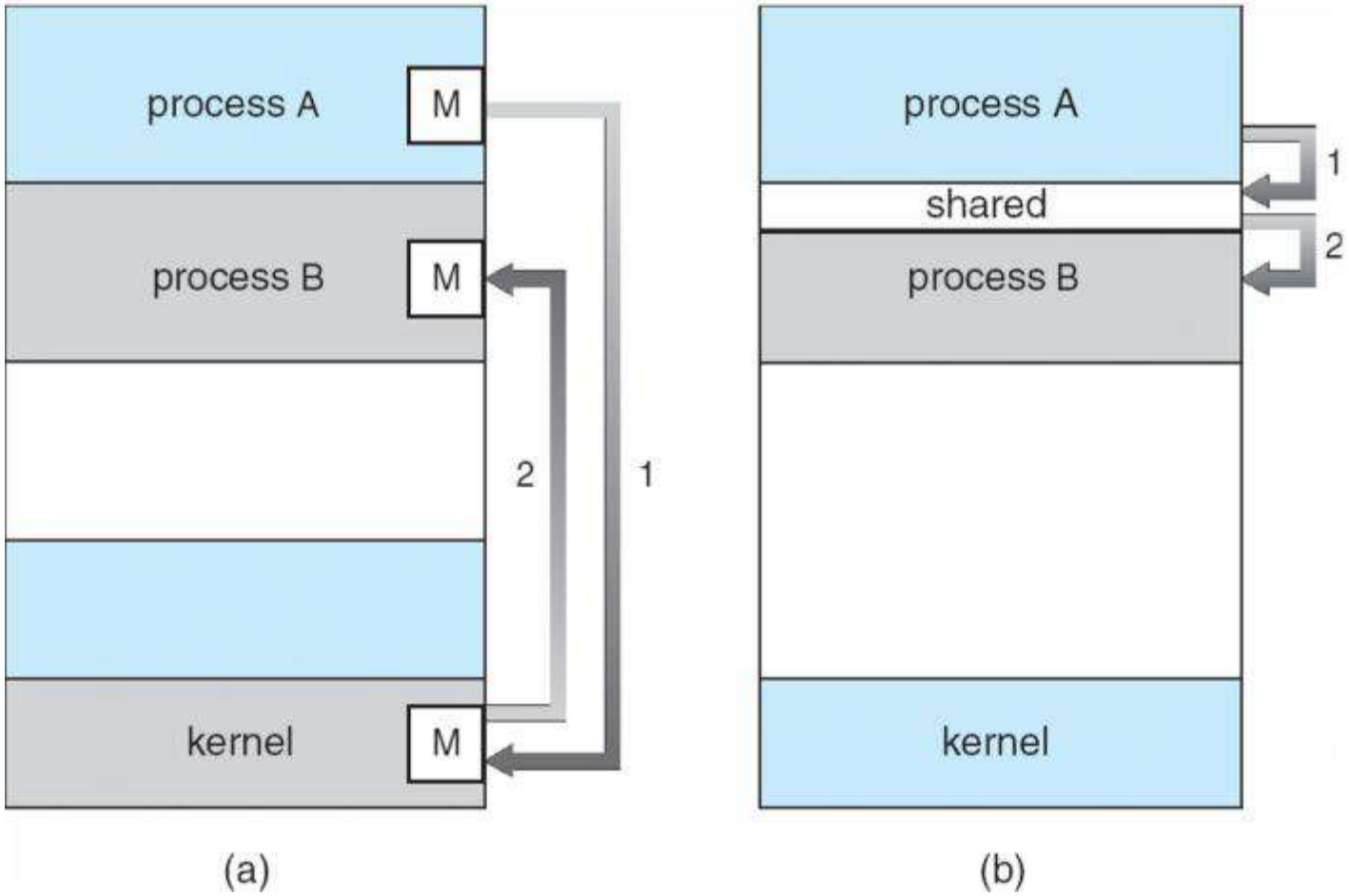
Process Termination

- Process executes last statement and asks the operating system to delete it (**exit)**
  - Output data from child to parent (via **wait)**
  - Process' resources are deallocated by operating system
- Parent may terminate execution of children processes (abort)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent is exiting
    - Some operating system do not allow child to continue if its parent terminates
    - –All children terminated -**cascading termination**

Interprocess Communication

- Processes within a system may be **independent or cooperating**
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need **interprocess communication (IPC)**
- Two models of IPC
  - Shared memory
  - Message passing

Communications Models



Cooperating Processes

- **Independent** process cannot affect or be affected by the execution of another process
- **Cooperating** process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

Signals

- A signal is a limited form of inter-process communication used in Unix, Unix-like, and other POSIX-compliant operating systems.
- It is an asynchronous notification sent to a process or to a specific thread within the same process in order to notify it of an event that occurred.
- When a signal is sent, the operating system interrupts the target process's normal flow of execution. Execution can be interrupted during any non-atomic instruction. If the process has previously registered a signal handler, that routine is executed. Otherwise the default signal handler is executed.
- Signals have been around since the 1970s Bell Labs Unix and are more recently specified in the POSIX standard.

Signals

- examples of Linux signal types:
  - SIGINT : interrupt from keyboard
  - SIGFPE : floating point exception
  - SIGKILL : terminate receiving process
  - SIGCHLD : child process stopped or terminated
  - SIGSEGV: segment access violation
- Signals are not presented to the process immediately when they are generated. They must wait until the process is running again.

- If a process has specified its own signal handler. The Kernel must call the signal handler The program counter is set to the signal handling routine and the parameters to the routine are added to the call frame or registers.

Handling signals

- Signal handlers can be installed with the signal() system call.
- If a signal handler is not installed for a particular signal, the default handler is used. Otherwise the signal is intercepted and the signal handler is invoked.
- The process can also specify two default behaviors, without creating a handler: ignore the signal (SIG_IGN) and use the default signal handler (SIG_DFL).
- There are two signals which cannot be intercepted and handled: SIGKILL and SIGSTOP.

Signals - example

```
#include <stdio.h>
#include <signal.h>
//#include <unistd.h>
int ctrl_c_count = 0;
void (* old_handler)(int);
void ctrl_c(int);
void main () {
     int c;
     old_handler = signal (SIGINT, ctrl_c );
     while ((c = getchar()) != '\n');
     printf("ctrl_c count = %d\n", ctrl_c_count);
     (void) signal (SIGINT, old_handler);

               for (;;);
}
void ctrl_c(int signum) {
     (void) signal (SIGINT, ctrl_c);                    // signals are automatically reset
     ++ctrl_c_count;
}    //see also the POSIX sigaction() call - more complex but better
```

Masking signals with sigprocmask()

- the (modern) "POSIX" function used to mask signals in the global context, is the sigprocmask() system call. It allows us to specify a set of signals to block, and returns the list of signals that were previously blocked. This is useful when we'll want to restore the previous masking state once we're done with our critical section.
- *Note*: each process on a unix system has its own signals mask, which is used by the operating system to specify which signals should be delivered to the proces, and which should be blocked. The sigprocmask system call is used to take a signals mask we created in user space, and update the one in stored in the kernel, using this user-space mask. The mask stored in the kernel is the one later considered by the operating system, when deciding whether to deliver a signal to the process, or block it.
- sigprocmask() accepts 3 parameters:
- int how defines if we want to add signals to the current process's mask (SIG_BLOCK), remove them from the current mask (SIG_UNBLOCK), or completely replace the current mask with the new mask (SIG_SETMASK).
- const sigset_t *setThe set of signals to be blocked, or to be added to the current mask, or removed from the current mask (depending on the 'how' parameter).
- sigset_t *oldsetIf this parameter is not NULL, then it'll contain the previous mask. We can later use this set to restore the situation back to how it was before we called sigprocmask().

Example – signals masking

```
void catch_int(int sig_num) {
sigset_t mask_set; /* used to set a signal masking set. */
sigset_t old_set; /* used to store the old mask set. */
 /* re-set the signal handler again to catch_int, for next time */
    signal(SIGINT, catch_int);
/* block any further signals while we're inside the handler. */
    sigfillset(&mask_set);
    sigprocmask(SIG_SETMASK, &mask_set, &old_set);
/// signal handling code
/* restore old mask */
sigprocmask(SIG_SETMASK, &old_set, &mask_set, );
```

Implementing Timers Using Signals

```
#include <unistd.h>
#include <signal.h>
 char user[40]; /* buffer to read user name from the user */ /* define an alarm signal handler. */
   void catch_alarm(int sig_num) {
     printf("Operation timed out. Exiting...\n\n"); exit(0); }
. . /* and inside the main program... */ . .
       /* set a signal handler for ALRM signals */
         signal(SIGALRM, catch_alarm);
         /* prompt the user for input */
          printf("Username: ");
         fflush(stdout);
          /* start a 30 seconds alarm */
          alarm(30);
         /* wait for user input */
         gets(user);
         /* remove the timer, now that we've got the user's input */
          alarm(0);
          . . /* do something with the received user name */
```

"Do" and "Don't" inside A Signal Handler

- Make it short
- Proper Signal Masking
- Careful with "fault" signals
- Careful with timers - only use one timer at a time
- Signals are NOT an event driven framework

How to send Signals to another process? Kill(..)

pipes

- Pipe is an effective way of communication between process. Pipe has descriptors. One descriptor is used for reading while other end is used for writing.

- Usage of pipe is to have communication between child and parent process. We also use pipe to redirect of output of a process to another process. We often use pipe in our shell scripts.

- Unidirectional, FIFO, unstructured data stream
- Fixed maximum size
- Simple flow control
- *pipe()* system call creates two file descriptors. Why?
- Implemented using filesystem, sockets or STREAMS (bidirectional pipe).

Implementation

- In most Unix-like systems, all processes of a pipeline are started at the same time, with their streams appropriately connected, and managed by the scheduler together with all other processes running on the machine.
- An important aspect of this, setting Unix pipes apart from other pipe implementations, is the concept of buffering: for example a sending program may produce 5000 bytes per second, and a receiving program may only be able to accept 100 bytes per second, but no data is lost. Instead, the output of the sending program is held in a queue.
- When the receiving program is ready to read data, the operating system sends its data from the queue, then removes that data from the queue. If the queue buffer fills up, the sending program is suspended (blocked) until the receiving program has had a chance to read some data and make room in the buffer. In Linux, the size of the buffer is 65536 bytes.

example- Using a pipe to send data from parent to a child

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[])
{
  int  f_des[2];
  static char message[BUFSIZ];

  if  ( argc != 2 ) {
    fprintf(stderr, "Usage: %s message\n", *argv);
    exit(1);
  }

  if ( pipe(f_des) == -1 ) {
    perror("Pipe");
    exit(2);
  }

  switch ( fork() ) {
  case -1:
    perror("Fork");
    exit(3);
```

Example- cont.

```
  case 0:                          /* In the child */
    close(f_des[1]);
    if ( read(f_des[0], message, BUFSIZ) != -1 ) {
      printf("Message received by child: [%s]\n", message);
      fflush(stdout);
    }
    else {
      perror("Read");
      exit(4);
    }
    break;

  default:                         /* In the parent */
    close(f_des[0]);
    if ( write(f_des[1], argv[1], strlen(argv[1])) != -1 ) {
      printf("Message sent by parent: [%s]\n", argv[1]);
      fflush(stdout);
    }
    else {
      perror("Write");
      exit(5);
    }
  }
  exit(0);
}
```

Named Pipes (FIFOs - First In First Out)

A named pipe works much like a regular pipe, but does have some noticeable differences.

- Named pipes exist as a device special file in the file system.
- Processes of different ancestry can share data through a named pipe.
- When all I/O is done by sharing processes, the named pipe remains in the file system for later use.

Creating a FIFO

- To create a FIFO in C, we can make use of the mknod() system call:
- PROTOTYPE:

int mknod(char *pathname, mode_t mode, dev_t dev);
RETURNS: 0 on success, -1 on error:

mknod("/tmp/MYFIFO", S_IFIFO|0666, 0);

FIFO Operations

- I/O operations on a FIFO are essentially the same as for normal pipes, with once major exception. An ``open'' system call or library function should be used to physically open up a channel to the pipe.

fifoserver.c

```
#include <stdio.h>
 #include <stdlib.h>
 #include <sys/stat.h>
#include <unistd.h>
#include <linux/stat.h>
 #define FIFO_FILE "MYFIFO"
int main(void) {
    FILE *fp;
    char readbuf[80]; /* Create the FIFO if it does not exist */
    umask(0);
    mknod(FIFO_FILE, S_IFIFO|0666, 0);
    while(1) {
        fp = fopen(FIFO_FILE, "r");
        fgets(readbuf, 80, fp);
                printf("Received string: %s\n", readbuf);
        fclose(fp); }
    return(0); }
```

fifoclient.c

```
#include <stdio.h>
 #include <stdlib.h>
 #define FIFO_FILE "MYFIFO"
 int main(int argc, char *argv[])
  { FILE *fp;
        if ( argc != 2 ) {
              printf("USAGE: fifoclient [string]\n");
        exit(1); }
        if((fp = fopen(FIFO_FILE, "w")) == NULL) {
              perror("fopen"); exit(1); }
        fputs(argv[1], fp);
        fclose(fp);
        return(0);
}
```

Blocking Actions on a FIFO

- Normally, blocking occurs on a FIFO. In other words, if the FIFO is opened for reading, the process will "block" until some other process opens it for writing. This action works vice-versa as well.
- If this behavior is undesirable, the O_NONBLOCK flag can be used in an open() call to disable the default blocking action.
- Can you think of another way to have a non-blocking FIFO?

Named Pipes –Win32

Here's a quick overview of the steps required to create and use a simple named pipe to send data from a server program to a client program.

Server program:

- Call **CreateNamedPipe(..)** to create an instance of a named pipe.
- Call **ConnectNamedPipe(..)** to wait for the client program to connect.
- Call **WriteFile(..)** to send data down the pipe.
- Call **CloseHandle(..)** to disconnect and close the pipe instance.

Named Pipes –Win32

Client program:

- Call **CreateFile(..)** to connect to the pipe.
- Call **ReadFile(..)** to get data from the pipe.
- Output data to the screen.
- Call **CloseHandle(..)** to disconnect from the pipe.

For more details visit

http://msdn.microsoft.com/en-us/library/windows/desktop/aa365590(v=vs.85).aspx

PIPE NAMES-WIN32

- You can name Win32 pipes almost anything you like, as long as they start with the prefix “\\.\pipe\”. In practice it becomes ”\\\\.\\pipe\\”
- Everything after that in the name is up to you, so long as you don’t use backslashes though
- don’t exceed 256 characters in total

READ/WRITE MODES

- There are two main modes of read/write operation available on pipes: byte stream, and message. The difference is fairly small, but can be very significant depending on your application.

- Message mode simply makes a distinction between each set of data sent down the pipe. If a program sends 50 bytes, then 100 bytes, then 40 bytes, the receiving program will receive it in these separate blocks (and will therefore need to read the pipe at least 3 times to receive everything).

- On the other hand, byte stream mode lets all the sent data flow continuously. In our example of 50, 100, then 40 bytes, the client could happily receive everything in a single 190-byte chunk. Which mode you choose depends on what your programs need to do

OVERLAPPED PIPE IO

- By default, pipe operations in Win32 are synchronous, or blocking. That means your program (or specifically the thread which handles the pipe operations) will need to wait for each operation to complete before it can continue.
- Using **overlapped pipe IO** means that pipe operations can process in the background while your program continues to do other things (including running other pipe operations in some cases). This can be very helpful, but it means you have to keep track of which operations are in progress, and monitor them for completion.
- An alternative to overlapped operation is to run synchronous pipe operations in a separate thread. If your pipe IO needs are fairly simple then this may be a simpler option.

BUFFERED INPUT/OUTPUT

- When calling “CreateNamedPipe(..)” you can choose to specify buffer sizes for outbound and inbound data.
- These can be very helpful for program performance, particularly in synchronous operation.
- If your buffer size is 0 (which is entirely valid) then every byte of data must be read from the other end of the pipe before the write operation can be completed
- However, if a buffer is specified then a certain amount of data can linger in the pipe before it gets read. This can allow the sending program to carry on with other tasks without needing to use overlapped pipe IO.

CreateNamedPipe

```
HANDLE hPipe;
hPipe = CreateNamedPipe(
    g_szPipeName, // pipe name
    PIPE_ACCESS_DUPLEX, // read/write access
    PIPE_TYPE_MESSAGE | // message type pipe
    PIPE_READMODE_MESSAGE | // message-read mode
    PIPE_WAIT, // blocking mode
    PIPE_UNLIMITED_INSTANCES, // max. instances
    BUFFER_SIZE, // output buffer size
    BUFFER_SIZE, // input buffer size
    NMPWAIT_USE_DEFAULT_WAIT, // client time-out
    NULL); // default security attribute
```

Message Passing

- In a *Message passing system* there are no shared variables. IPC facility provides two operations for fixed or variable sized message:
  - *send(message)*
  - *receive(message)*

- If processes *P* and *Q* wish to communicate, they need to:
  - establish a *communication link*
  - exchange messages via *send and receive*

- Implementation of communication link
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., syntax and semantics, abstractions)

Implementation Questions

- How are links established?
- Can a link be associated with more than two processes?
- How are links made known to processes?
- How many links can there be between every pair/group of communicating processes?
- What is the capacity of a link?
- Is the size of a message that the link can accommodate fixed or variable?
- Is a link unidirectional or bi-directional?

Message Passing Systems

- Exchange messages over a communication link

- Methods for implementing the communication link and primitives (*send*/*receive*):
  1. Direct or Indirect communications (*Naming*)
  2. Symmetric or Asymmetric communications
  3. Automatic or Explicit buffering
  4. Send-by-copy or send-by-reference
  5. fixed or variable sized messages

Message Queues

- Message queues can be best described as an internal linked list within the kernel's addressing space. Messages can be sent to the queue in order and retrieved from the queue in several different ways. Each message queue (of course) is uniquely identified by an IPC identifier.

Internal and User Data Structures

- **Message buffer :** the msgbuf structurebe thought of as a *template* for message data.
- **Kernel msg structure :** message details
- **Kernel msqid_ds structure:** queue details
- **Kernel ipc_perm structure:** permission information for IPC objects.

Message buffer

- The msgbuf structure can be thought of as a *template* for message data.

```
struct msgbuf {
long mtype; /* type of message */
char mtext[1]; /* message data*/
};
```

- It is up to the programmer to define structures of this type.

```
struct my_msgbuf {
long mtype; /* Message type */
long request_id; /* Request identifier */
 struct client info; /* Client information structure */
 };
```

SYSTEM CALL: msgget()

- In order to create a new message queue, or access an existing queue, the msgget() system call is used
- int msgget (key_t key, int msgflg);
  - The first argument to msgget() is the key value (in our case returned by a call to ftok()).
  - the msgflg argument
    - **IPC_CREAT:** Create the queue if it doesn't already exist in the kernel.
    - **IPC_EXCL:** When used with IPC_CREAT, fail if queue already exists.
- Msgget() returns the message queue identifier

Example- open_queue()

```
int open_queue( key_t keyval )
 {
    int qid;
    if((qid = msgget( keyval, IPC_CREAT | 0660 )) == -1)
          {
                return(-1);
          }
    return(qid);
}
```

SYSTEM CALL: msgsnd()

- Once we have the queue identifier, we can begin performing operations on it. To deliver a message to a queue, you use the msgsnd system call:
- **int msgsnd (int msqid, struct msgbuf *msgp, int msgsz, int msgflg);**
  - RETURNS: 0 on success -1 on error
  - msgsnd: queue identifier
  - msgp: pointer to the message buffer.
  - Msgsz: size of the message in bytes, excluding the length of the message type (4 byte long).
  - msgflg : 0 (ignored), or:
    - **IPC_NOWAIT** If the message queue is full, then the message is not written to the queue.

Example- send_message

```
int send_message( int qid, struct mymsgbuf *qbuf )
 {
     int result, length;
    /* The length is essentially the size of the structure minus
    sizeof(mtype) */
    length = sizeof(struct mymsgbuf) - sizeof(long);
     if((result = msgsnd( qid, qbuf, length, 0)) == -1)
          {
                return(-1);
          }
    return(result);
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <linux/ipc.h>
 #include <linux/msg.h>
main() {
    int qid;
    key_t msgkey;
    struct mymsgbuf {
            long mtype; /* Message type */
            int request; /* Work request number */
            double salary; /* Employee's salary */ } msg;
    /* Generate our IPC key value */
    msgkey = ftok(".", 'm');
    /* Open/create the queue */
    if(( qid = open_queue( msgkey)) == -1)
     { perror("open_queue"); exit(1); }
    /* Load up the message with arbitrary test data */
    msg.mtype = 1; /* Message type must be a positive number! */
    msg.request = 1; /* Data element #1 */
    msg.salary = 1000.00; /* Data element #2 (my yearly salary!)*/
if((send_message( qid, &msg )) == -1)
    { perror("send_message"); exit(1); } }
```

SYSTEM CALL: msgrcv()

- Retrieving the message from the queue is done using the msgrcv() system call:
- **int msgrcv (int msqid, struct msgbuf *msgp, int msgsz, long mtype, int msgflg);**
  - RETURNS: Number of bytes copied into message buffer
  - **msqid,msgp and msgsz** same as **msgsnd()**
  - **mtype**: specifies the *type* of message to retrieve from the queue
  - **msgflg: IPC_NOWAIT** is passed as a flag, and no messages are available, the call returns ENOMSG to the calling process. Otherwise, the calling process blocks

Example- read_message()

```
int read_message( int qid, long type, struct mymsgbuf *qbuf )
 {
    int result, length;
/* The length is essentially the size of the structure minus
    sizeof(mtype) */
    length = sizeof(struct mymsgbuf) - sizeof(long);
    If((result = msgrcv( qid, qbuf, length, type, 0)) == -1)
          { return(-1); }
    return(result);
}
```

Example- peek_message()

```
int peek_message( int qid, long type )
   {
        int result, length;
        if((result = msgrcv( qid, NULL, 0, type, IPC_NOWAIT)) == -1)
                {
                        if(errno == E2BIG)
                                return(TRUE);
                }
        return(FALSE);
   }
```

SYSTEM CALL: msgctl()

- To perform control operations on a message queue, you use the msgctl() system call.
- int msgctl(int msgqid, int cmd, struct msqid_ds *buf);
  - RETURNS: 0 on success
  - cmd:
    - **IPC_STAT:** Retrieves the **msqid_ds** structure for a queue, and stores it in the address of the buf argument.
    - **IPC_SET:** Sets the value of the **ipc_perm** member of the msqid_ds structure for a queue. Takes the values from the buf argument.
    - **IPC_RMID:** Removes the queue from the kernel.

Example- remove_queue

```
int remove_queue( int qid )
   {
        if( msgctl( qid, IPC_RMID, 0) == -1)
                {
                        return(-1);
                }
         return(0);
   }
```

Example- Server

```
#include…..

#define KEY 500
struct msg
{
      long int type;
      char a[1024];
      int pid;
      }p,p1;
int main()
{
      int m;
      m=msgget(KEY,0666|IPC_CREAT);
      p.type=1;
      printf("\nEnter the msg");
      scanf("%s",&p.a);
      pid_t pid;
      p.pid=getpid();
      msgsnd(m,&p,sizeof(p),0);
      msgrcv(m,&p1,sizeof(p),p.pid,0);
      printf("%s",p1.a);
}
```

Example- Client

```
#define KEY 500
struct msg
{
    long int type;
    char a[1024];
    int pid;
}p;
int main()
    {
        int m,n,fd,m1;
        m=msgget(KEY,0666|IPC_CREAT);
        while(1)
        {
            msgrcv(m,&p,sizeof(p),1,0);
            printf("Filename from client %s\n",p.a);
            fd=open(p.a,O_RDONLY);
            n=read(fd,p.a,1024);
            p.type=p.pid;
            p.pid=getpid();
            msgsnd(m,&p,sizeof(p),0);
        } }
```

Example- Output

```
Enter the msg strcmp.c
Filename from client strcmp.c
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<stdlib.h>
#include<string.h>
int main(int argc,char *argv[])
{
	if(strcmp(argv[1],argv[2])==0)
		printf("The given strings are equal");
	else
		printf("The strings are not equal");
}
```

Mailslot – Win32

- Mailslot is used for one way inter-process communications.
- There is a Mailslot server which will be read-only; it will just read the client sent messages.
- The clients will be write-only clients, sending messages to the server.
- Mailslot messages can be of around 400 bytes only.
- Mailslot can broadcast messages in a domain. If processes in a domain create a mailslot with the same name, then a message that is sent to that mailslot is sent to all of these processes.

Mailslot Win32 APIs

- Following are some of the Win32 APIs that are used when working with Mailslot:
  - CreateMailSlot()
  - GetMailslotInfo()
  - SetMailslotInfo()
  - ReadFile()
  - WriteFile()
  - CloseHandle()

Mailslot name

- A Mailslot name needs to be in the following format:
  - *ComputerName*\mailslot\[path\]name
  - *DomainName*\mailslot\[path\]name
  - *\mailslot\[path\]name

Demo

Shared Memory

- Shared memory can best be described as the mapping of an area (segment) of memory that will be mapped and shared by more than one process.
- This is by far the fastest form of IPC.
- information is mapped directly from a memory segment, and into the addressing space of the calling process
- A segment can be created by one process, and subsequently written to and read from by any number of processes.

Shared memory - unix

- the kernel maintains a special internal data structure for each shared memory segment which exists within its addressing space. This structure is of type **shmid_ds**, and is defined in linux/shm.h
- The shared memory is managed using four system calls
  - shmget()
  - shmat()
  - shmctl()
  - shmdt()

SYSTEM CALL: shmget()

- In order to create a new shared memory, or access an existing one, the shmget() system call is used.
- int shmget (key_t key, int size, int shmflg);
  - RETURNS: shared memory segment identifier on success -1 on error
  - key :the IPC key value (in our case returned by a call to ftok()).
  - the msgflg argument
    - **IPC_CREAT:** Create the segment if it doesn't already exist in the kernel.
    - **IPC_EXCL:** When used with IPC_CREAT, fail if segment already exists.

Example- open_segment

```
int open_segment( key_t keyval, int segsize )
{
    int shmid;
    if((shmid = shmget( keyval, segsize, IPC_CREAT | 0660 )) == -1)
        {
            return(-1);
        }
    return(shmid);
}
```

SYSTEM CALL: shmat()

- Once a process has a valid IPC identifier for a given segment, the next step is for the process to attach or map the segment into its own addressing space
- int shmat (int shmid, char *shmaddr, int shmflg);
  - RETURNS: address at which segment was attached to the process, or -1 on error
  - If the addr argument is zero (0), the kernel tries to find an unmapped region
  - if the SHM_RDONLY flag is OR'd in with the flag argument, then the shared memory segment will be mapped in, but marked as readonly.

Example- attach_segment

```
char *attach_segment( int shmid )
   {
        return(shmat(shmid, 0, 0));
   }
```

SYSTEM CALL: shmctl()

int shmctl (int shmqid, int cmd, struct shmid_ds *buf);

- Commands:
  - **IPC_STAT** Retrieves the shmid_ds structure for a segment, and stores it in the address of the buf argument
  - **IPC_SET** Sets the value of the ipc_perm member of the shmid_ds structure for a segment. Takes the values from the buf argument.
  - **IPC_RMID** Marks a segment for removal.
- The IPC_RMID command doesn't actually remove a segment from the kernel. Rather, it marks the segment for removal. The actual removal itself occurs when the last process currently attached to the segment has properly detached it. Of course, if no processes are currently attached to the segment, the removal seems immediate.

SYSTEM CALL: shmdt()

int shmdt (char *shmaddr);

- After a shared memory segment is no longer needed by a process, it should be detached by calling this system call.
- this is not the same as removing the segment from the kernel! After a detach is successful, the **shm_nattch** member of the associates **shmid_ds** structure is decremented by one. When this value reaches zero (0), the kernel will physically remove the segment.

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;
key = 5678; /* * Create the segment. */
if ((shmid = shmget(key, SHMSZ, IPC_CREAT | 0666)) < 0)
{ perror("shmget"); exit(1); }
/* * Now we attach the segment to our data space. */
 if ((shm = shmat(shmid, NULL, 0)) == (char *) -1)
{ perror("shmat"); exit(1); }
/* * Now put some things into the memory for the * other process to read. */
    s = shm;
   for (c = 'a'; c <= 'z'; c++) *s++ = c; *s = NULL;
 /* * Finally, we 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() {
    char c; int shmid; key_t key;
    char *shm, *s;
    key = 5678; /* * Locate the segment.. */
    if ((shmid = shmget(key, SHMSZ, IPC_CREAT | 0666)) < 0)
        { perror("shmget"); exit(1); }
/* * Now we attach the segment to our data space. */
    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1)
        { perror("shmat"); exit(1); }
/* * Now 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); }
```

Memory Mapped files – win32

- File mapping is an efficient way for two or more processes on the same computer to share data.
- In order to access the file's contents, the processes uses virtual address space called file view.
- Processes read from and write to the file view using pointers, just as they would with dynamically allocated memory.

Shared memory win32 APIs

- Following are some of the Win32 APIs that are used when working with shared memory (memory mapped objects):
  - CreateFileMapping() : create shared memory
  - MapViewOfFile() : attach the process to the shared segment
  - UnMapViewOfFile() detach the process from the shared segment
  - CloseHandle()

CreateFileMapping()

HANDLE CreateFileMapping(

HANDLE *hFile*,// handle to file to map -- INVALID_HANDLE_VALUE, if only shared memory
LPSECURITY_ATTRIBUTES *lpFileMappingAttributes,*

DWORD *flProtect*, // protection for mapping object
DWORD *dwMaximumSizeHigh*,

DWORD *dwMaximumSizeLow,*

LPCTSTR *lpName*// name of file-mapping object**);**

- The flProtect argument can be one of the following.
  - PAGE_READONLY - memory is readonly
  - PAGE_READWRITE - memory is readable and writable
  - PAGE_WRITECOPY - memory is readable, after memory has been written too.

MapViewOfFile()

LPVOID MapViewOfFile(

HANDLE *hFileMappingObject*,// file-mapping object
DWORD *dwDesiredAccess*,// access mode
DWORD *dwFileOffsetHigh*,// file offset
DWORD *dwFileOffsetLow*,// file offset
DWORD *dwNumberOfBytesToMap*);

- Function maps a buffer referred to by the file mapping object to the local process space of the current process. The function returns NULL, if there was an error. Otherwise, it returns a legal address. The dwDesiredAccess has the following values:
  - FILE_MAP_READ - mapping can only be read only
  - FILE_MAP_WRITE - mapping can either be read or write

UnMapViewOfFile()

- A file buffer that was mapped to the current process space is realeased with this function. The lpBaseAddress argument takes the return value from the **MapViewOfFile** function. The return value of this function is TRUE on success.
- **BOOL UnmapViewOfFile(LPCVOID** *lpBaseAddress***);**

Example

```
struct TSharedMemory {
  DWORD m_dwProcessID;  // Process ID from client
  CHAR m_cText[512];   // Text from client to server
  UINT m_nTextLength;      // Returned from client
};
m_hMap = ::CreateFileMapping(
        (HANDLE) INVALID_HANDLE_VALUE,
         NULL,
        PAGE_READWRITE,
        0,
         sizeof(TSharedMemory),
        “ApplicationSpecificSharedMem");
        m_pMsg =
            (TSharedMemory*)::MapViewOfFile(m_hMap,FILE_MAP_WRITE,0,0,sizeof(TS
            haredMemory));
        // read and write to the memory
        ::CloseHandle(m_hMap);
```

Semaphores

- Semaphores can best be described as counters used to control access to shared resources by multiple processes. They are most often used as a locking mechanism to prevent processes from accessing a particular resource while another process is performing operations on it.
- think of them as *resource counters*
- it is important to realize that semaphores are actually implemented as *sets (Unix)*, rather than as single entities.

Semaphores- Unix

- As with message queues, the kernel maintains a special internal data structure for each semaphore set which exists within its addressing space. This structure is of type **semid_ds**
- **sem** structure exists for every semaphore in the set and contains information like the current count

SYSTEM CALL: semget()

- int semget (key_t key, int nsems, int semflg);
  - RETURNS: semaphore set IPC identifier on success -1 on error
  - key :the IPC key value (in our case returned by a call to ftok()).
  - the msgflg argument
    - **IPC_CREAT:** Create the semaphore set if it doesn't already exist in the kernel.
    - **IPC_EXCL:** When used with IPC_CREAT, fail if semaphore already exists.

Example - open_semaphore_set

```
int open_semaphore_set( key_t keyval, int numsems )
 {
   int sid;
    if ( ! numsems )
         return(-1);
   if((sid = semget( mykey, numsems, IPC_CREAT | 0660 )) == -1)
         { return(-1); }
   return(sid);
 }
```

SYSTEM CALL: semop()

- int semop (int semid, struct sembuf *sops, unsigned nsops);
  - RETURNS: 0 on success (all operations performed) -1 on error
  - The first argument to semget() is the key value (in our case returned by a call to semget).
  - The second argument (sops) is a pointer to an array of *operations* to be performed on the semaphore set
  - the third argument (nsops) is the number of operations in that array.

```
struct sembuf {
    ushort sem_num; /* semaphore index in array */ short sem_op; /*
    semaphore operation */ short sem_flg; /* operation flags */
 };
```

- If sem_op is negative, then its value is subtracted from the semaphore. This correlates with obtaining resources that the semaphore controls
- If sem_op is positive, then it's value is added to the semaphore. This correlates with returning resources back to the application's semaphore set
- if sem_op is zero (0), then the calling process will sleep() until the semaphore's value is 0. This correlates to waiting for a semaphore to reach 100% utilization.

example

```
struct sembuf acquire = {0, -1, SEM_UNDO},
              release = {0,  1, SEM_UNDO};
semid = semget(ipc_key, 2, IPC_CREAT |  IPC_EXCL | 0660)) ;
semop(semid, &acquire, 1);
/// use the shared recource
semop(semid, &release, 1);
```

SYSTEM CALL: semctl()

int semctl (int semid, int semnum, int cmd, union semun arg);

- **IPC_STAT** Retrieves the semid_ds structure for a set, and stores it in the address of the buf argument in the semun union.
- **IPC_SET** Sets the value of the ipc_perm member of the semid_ds structure for a set. Takes the values from the buf argument of the semun union.
- **IPC_RMID** Removes the set from the kernel.
- **GETALL** Used to obtain the values of all semaphores in a set. The integer values are stored in an array of unsigned short integers pointed to by the *array* member of the union.
- **GETNCNT** Returns the number of processes currently waiting for resources.
- **GETPID** Returns the PID of the process which performed the last *semop* call.
- **GETVAL** Returns the value of a single semaphore within the set.
- **GETZCNT** Returns the number of processes currently waiting for 100% resource utilization.
- **SETALL** Sets all semaphore values with a set to the matching values contained in the *array* member of the union.
- **SETVAL** Sets the value of an individual semaphore within the set to the *val* member of the union.

```
union semun {
      int val; /* value for SETVAL */
     struct semid_ds *buf; // buffer for IPC_STAT & IPC_SET
     ushort *array; // array for GETALL & SETALL
     struct seminfo *__buf; // buffer for IPC_INFO
     void *__pad; };

int get_sem_val( int sid, int semnum )
{
     return( semctl(sid, semnum, GETVAL, 0));
}

#define MAX_PRINTERS 5
printer_usage()
{
     int x;
     for(x=0; x<MAX_PRINTERS; x++)
     printf("Printer %d: %d\n\r", x, get_sem_val( sid, x ));
}
```

- Consider the following function, which could be used to initialize a new semaphore value:

```
void init_semaphore( int sid, int semnum, int initval)
   {
        union semun semopts;
        semopts.val = initval;
        semctl( sid, semnum, SETVAL, semopts);
   }
```

Semaphores – Win32

- CreateSemaphore( ) : create new or open existing semaphore
- WaitForSingleObject(): acquire semaphore
- ReleaseSemaphore(): release semaphore
- CloseHandle()

CreateSemaphore()

```
HANDLE CreateSemaphore(
    LPSECURITY_ATTRIBUTES
    lpSemaphoreAttributes, LONG lInitialCount,
    LONG lMaximumCount, LPCTSTR lpName );
```

WaitForSingleObject()

```
DWORD WINAPI WaitForSingleObject(
  HANDLE hHandle,
  DWORD dwMilliseconds );
```

ReleaseSemaphore

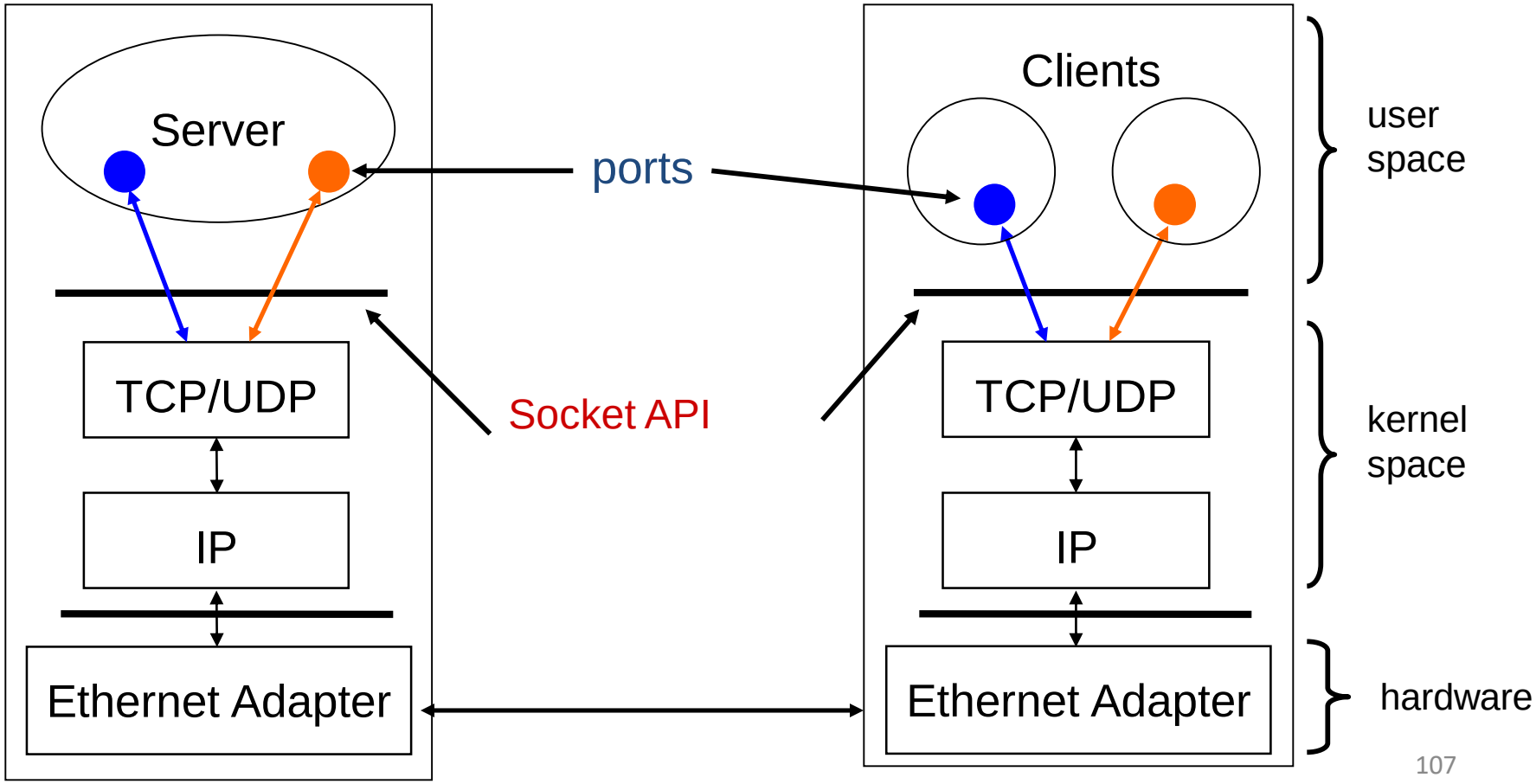
```
BOOL WINAPI ReleaseSemaphore(
    HANDLE hSemaphore,
    LONG lReleaseCount,
    LPLONG lpPreviousCount );
```

- ***lReleaseCount*** The amount by which the semaphore object's current count is to be increased. The value must be greater than zero. If the specified amount would cause the semaphore's count to exceed the maximum count that was specified when the semaphore was created, the count is not changed and the function returns **FALSE**.

Introduction to Socket Programming

Server and Client

Server and Client exchange messages over the network through a common Socket API



User Datagram Protocol(UDP): An Analogy

UDP

- Single socket to receive messages
- No guarantee of delivery
- Not necessarily in-order delivery
- Datagram – independent packets
- Must address each packet

Postal Mail

- Single mailbox to receive letters
- Unreliable ☺
- Not necessarily in-order delivery
- Letters sent independently
- Must address each reply

Example UDP applications
Multimedia, voice over IP

Transmission Control Protocol (TCP): An Analogy

TCP

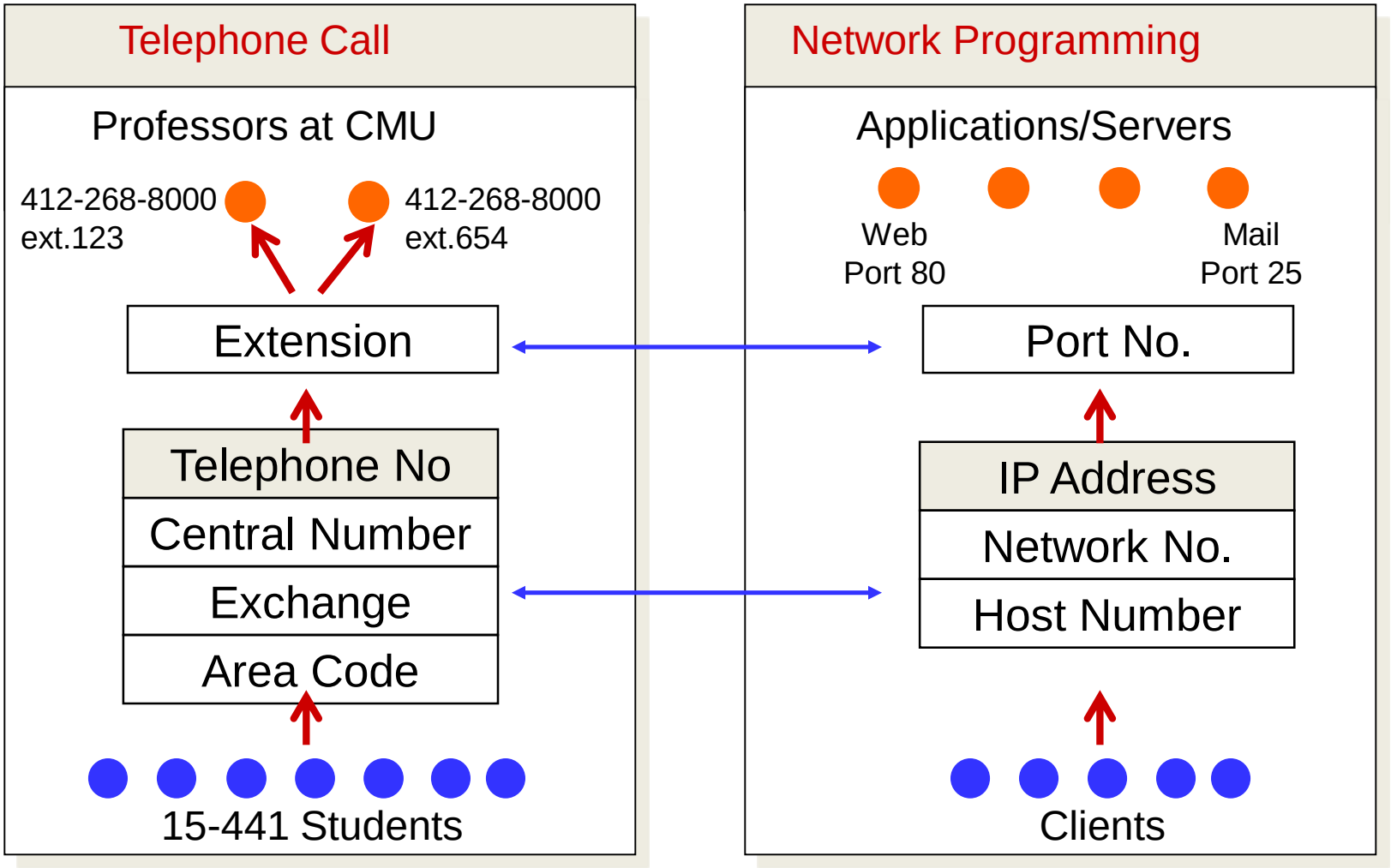
- Reliable – guarantee delivery
- Byte stream – in-order delivery
- Connection-oriented – single socket per connection
- Setup connection followed by data transfer

Telephone Call

- Guaranteed delivery
- In-order delivery
- Connection-oriented
- Setup connection followed by conversation

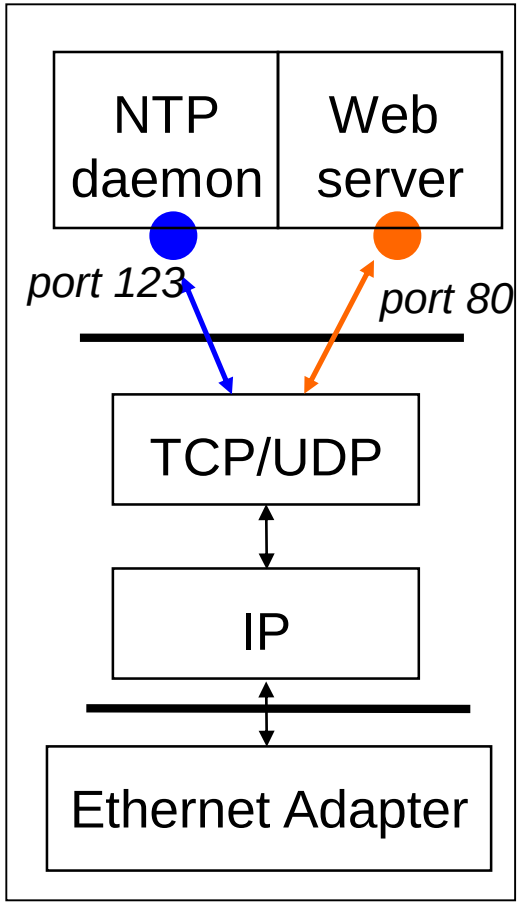
Example TCP applications
Web, Email, Telnet

Network Addressing Analogy



Concept of Port Numbers

- Port numbers are used to identify "entities" on a host
- Port numbers can be
  - Well-known (port 0-1023)
  - Dynamic or private (port 1024-65535)
- Servers/daemons usually use well-known ports
  - Any client can identify the server/service
  - HTTP = 80, FTP = 21, Telnet = 23, ...
  - /etc/service defines well-known ports
- Clients usually use dynamic ports
  - Assigned by the kernel at run time



Names and Addresses

- Each attachment point on Internet is given unique address
  - Based on location within network – like phone numbers
- Humans prefer to deal with names not addresses
  - DNS provides mapping of name to address
  - Name based on administrative ownership of host

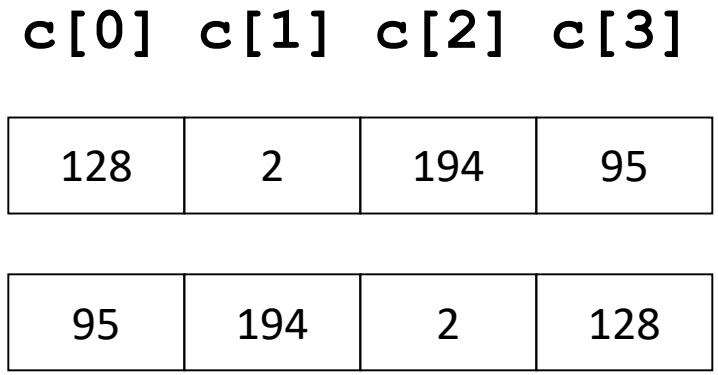
Internet Addressing Data Structure

```
#include <netinet/in.h>

/* Internet address structure */
struct in_addr {
        u_long s_addr;          /* 32-bit IPv4 address */
};                              /* network byte ordered */

/* Socket address, Internet style. */
struct sockaddr_in {
      u_char  sin_family;       /* Address Family */
      u_short sin_port;         /* UDP or TCP Port# */
                                /* network byte ordered */
      struct in_addr sin_addr;  /* Internet Address */
      char    sin_zero[8];      /* unused */
};
```

- sin_family = AF_INET selects Internet address family

Byte Ordering

```
union {
    u_int32_t addr;  /* 4 bytes address */
    char c[4];
} un;
/* 128.2.194.95 */
un.addr = 0x8002c25f;
/* c[0] = ? */
```



- Big Endian
  - Sun Solaris, PowerPC, ...
- Little Endian
  - i386, alpha, ...
- Network byte order = Big Endian

Byte Ordering Functions

- Converts between **host byte order** and **network byte order**
  - 'h' = host byte order
  - 'n' = network byte order
  - 'l' = long (4 bytes), converts IP addresses
  - 's' = short (2 bytes), converts port numbers

```
#include <netinet/in.h>

unsigned long int htonl(unsigned long int hostlong);
unsigned short int htons(unsigned short int
hostshort);
unsigned long int ntohl(unsigned long int netlong);
unsigned short int ntohs(unsigned short int
netshort);
```

Lecture Overview

- Background
  - TCP vs. UDP
  - Byte ordering
- Socket I/O
  - TCP/UDP server and client
  - I/O multiplexing

What is a Socket?

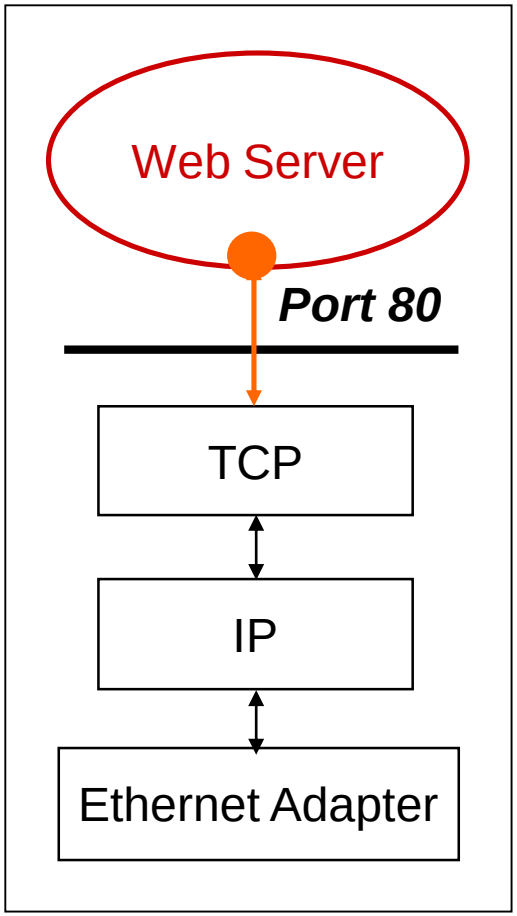
- A socket is a file descriptor that lets an application read/write data from/to the network

```
int fd;          /* socket descriptor */
if ((fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
      perror("socket");
      exit(1);
}
```

- ***socket*** returns an integer (socket descriptor)
  - fd < 0 indicates that an error occurred
  - socket descriptors are similar to file descriptors

- AF_INET: associates a socket with the Internet protocol family
- SOCK_STREAM: selects the TCP protocol
- SOCK_DGRAM: selects the UDP protocol

TCP Server



- For example: web server
- **What does a *web server* need to do so that a *web client* can connect to it?**

Socket I/O: socket()

- Since web traffic uses TCP, the web server must create a socket of type SOCK_STREAM

```
int fd;             /* socket descriptor */

if((fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
      perror("socket");
      exit(1);
}
```

- ***socket*** returns an integer (**socket descriptor**)
  - ***fd*** < 0 indicates that an error occurred
- **AF_INET** associates a socket with the Internet protocol family
- **SOCK_STREAM selects the TCP protocol**

Socket I/O: bind()

- A ***socket*** can be bound to a ***port***

```
int fd;                                  /* socket descriptor */
struct sockaddr_in srv;     /* used by bind() */

/* create the socket */

srv.sin_family = AF_INET; /* use the Internet addr family */

srv.sin_port = htons(80); /* bind socket 'fd' to port 80*/

/* bind: a client may connect to any of my addresses */
srv.sin_addr.s_addr = htonl(INADDR_ANY);

if(bind(fd, (struct sockaddr*) &srv, sizeof(srv)) < 0) {
       perror("bind"); exit(1);
}
```

- **Still not quite ready to communicate with a client...**

Socket I/O: listen()

- ***listen*** indicates that the server will accept a connection

```
int fd;                          /* socket descriptor */
struct sockaddr_in srv;          /* used by bind() */

/* 1) create the socket */
/* 2) bind the socket to a port */

if(listen(fd, 5) < 0) {
      perror("listen");
      exit(1);
}
```

- **Still not quite ready to communicate with a client...**

Socket I/O: accept()

- ***accept*** blocks waiting for a connection

```
int fd;                                  /* socket descriptor */
struct sockaddr_in srv;                 /* used by bind() */
struct sockaddr_in cli;                 /* used by accept() */
int newfd;                              /* returned by accept() */
int cli_len = sizeof(cli); /* used by accept() */

/* 1) create the socket */
/* 2) bind the socket to a port */
/* 3) listen on the socket */

newfd = accept(fd, (struct sockaddr*) &cli, &cli_len);
if(newfd < 0) {
       perror("accept");   exit(1);
}
```

- ***accept*** returns a new socket (***newfd***) with the same properties as the original socket (***fd***)
  - ***newfd*** < 0 indicates that an error occurred

Socket I/O: accept() continued…

```
struct sockaddr_in cli;          /* used by accept() */
int newfd;                       /* returned by accept() */
int cli_len = sizeof(cli);       /* used by accept() */

newfd = accept(fd, (struct sockaddr*) &cli, &cli_len);
if(newfd < 0) {
      perror("accept");
      exit(1);
}
```

- How does the server know which client it is?
  - **cli.sin_addr.s_addr** contains the client's ***IP address***
  - **cli.sin_port** contains the client's ***port number***
- Now the server can exchange data with the client by using ***read*** and ***write*** on the descriptor ***newfd***.
- Why does ***accept*** need to return a new descriptor?

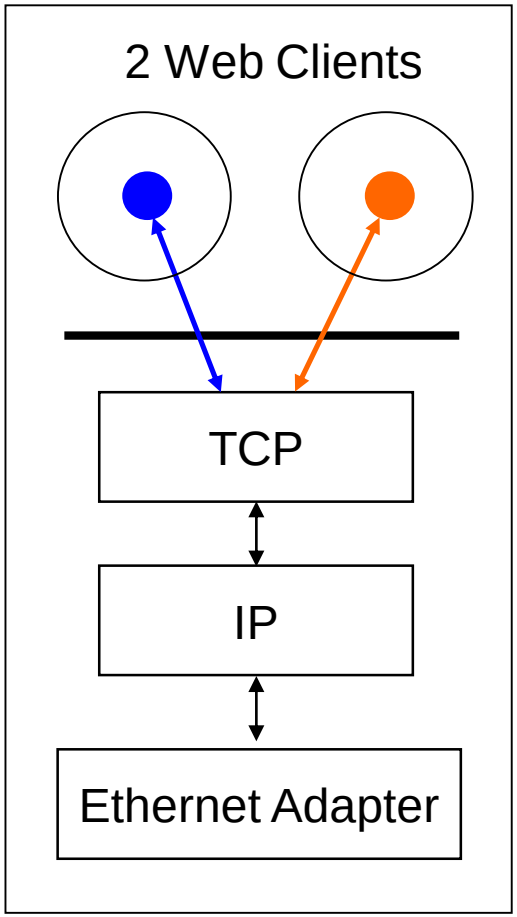
Socket I/O: read()

- ***read*** can be used with a socket
- ***read* blocks waiting for data from the client but does not guarantee that sizeof(buf) is read**

```
int fd;                                  /* socket descriptor */
char buf[512];                           /* used by read() */
int nbytes;                             /* used by read() */

/* 1) create the socket */
/* 2) bind the socket to a port */
/* 3) listen on the socket */
/* 4) accept the incoming connection */

if((nbytes = read(newfd, buf, sizeof(buf))) < 0) {
       perror("read"); exit(1);
}
```

TCP Client

- For example: web client
- **How does a *web client* connect to a *web server*?**



Dealing with IP Addresses

- IP Addresses are commonly written as strings (“128.2.35.50”), but programs deal with IP addresses as integers.

Converting strings to numerical address:

```
struct sockaddr_in srv;

srv.sin_addr.s_addr = inet_addr(“128.2.35.50”);
if(srv.sin_addr.s_addr == (in_addr_t) -1) {
      fprintf(stderr, "inet_addr failed!\n"); exit(1);
}
```

Converting a numerical address to a string:

```
struct sockaddr_in srv;
char *t = inet_ntoa(srv.sin_addr);
if(t == 0) {
      fprintf(stderr, “inet_ntoa failed!\n”); exit(1);
}
```

Translating Names to Addresses

- Gethostbyname provides interface to DNS
- Additional useful calls
  - Gethostbyaddr – returns `hostent` given sockaddr_in
  - Getservbyname
    - Used to get service description (typically port number)
    - Returns `servent` based on name

```
#include <netdb.h>

struct hostent *hp; /*ptr to host info for remote*/
struct sockaddr_in peeraddr;
char *name = "www.cs.cmu.edu";

peeraddr.sin_family = AF_INET;
hp = gethostbyname(name)
peeraddr.sin_addr.s_addr = ((struct in_addr*)(hp->h_addr))->s_addr;
```

Socket I/O: connect()

- ***connect*** allows a client to connect to a server...

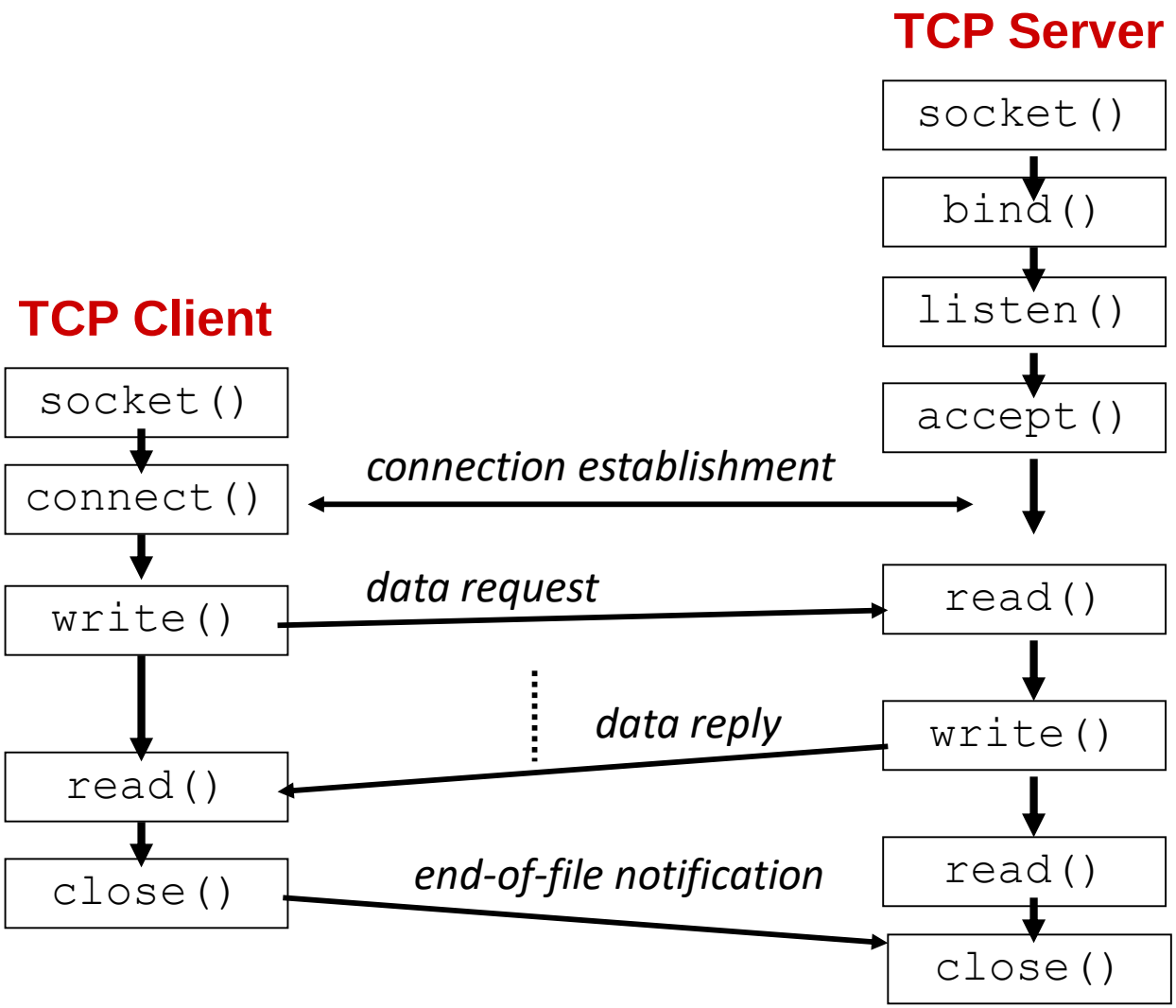
```
int fd;                                 /* socket descriptor */
struct sockaddr_in srv;                 /* used by connect() */

/* create the socket */

/* connect: use the Internet address family */
srv.sin_family = AF_INET;

/* connect: socket 'fd' to port 80 */
srv.sin_port = htons(80);

/* connect: connect to IP Address "128.2.35.50" */
srv.sin_addr.s_addr = inet_addr("128.2.35.50");

if(connect(fd, (struct sockaddr*) &srv, sizeof(srv)) < 0) {
        perror("connect"); exit(1);
}
```

Socket I/O: write()

- ***write*** can be used with a socket

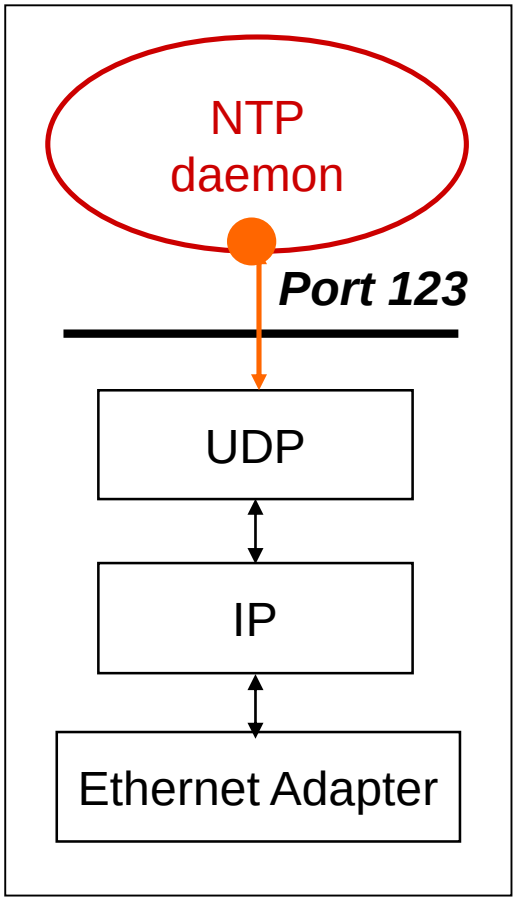
```
int fd;                          /* socket descriptor */
struct sockaddr_in srv;          /* used by connect() */
char buf[512];                   /* used by write() */
int nbytes;                      /* used by write() */

/* 1) create the socket */
/* 2) connect() to the server */

/* Example: A client could "write" a request to a server
*/
if((nbytes = write(fd, buf, sizeof(buf))) < 0) {
      perror("write");
      exit(1);
}
```

Review: TCP Client-Server Interaction



from UNIX Network Programming Volume 1, figure 4.1

UDP Server Example



- For example: NTP daemon

- **What does a *UDP server* need to do so that a *UDP client* can connect to it?**

Socket I/O: socket()

- The UDP server must create a **datagram** socket…

```
int fd;             /* socket descriptor */

if((fd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
      perror("socket");
      exit(1);
}
```

- ***socket*** returns an integer (**socket descriptor**)
  - ***fd*** < 0 indicates that an error occurred

- AF_INET: associates a socket with the Internet protocol family
- **SOCK_DGRAM: selects the UDP protocol**

Socket I/O: bind()

- A ***socket*** can be bound to a ***port***

```
int fd;                                 /* socket descriptor */
struct sockaddr_in srv;     /* used by bind() */

/* create the socket */

/* bind: use the Internet address family */
srv.sin_family = AF_INET;

/* bind: socket 'fd' to port 80*/
srv.sin_port = htons(80);

/* bind: a client may connect to any of my addresses */
srv.sin_addr.s_addr = htonl(INADDR_ANY);

if(bind(fd, (struct sockaddr*) &srv, sizeof(srv)) < 0) {
        perror("bind"); exit(1);
}
```

- **Now the UDP server is ready to accept packets…**

Socket I/O: recvfrom()

- ***read*** does not provide the client's address to the UDP server

```
int fd;                              /* socket descriptor */
struct sockaddr_in srv;              /* used by bind() */
struct sockaddr_in cli;              /* used by recvfrom() */
char buf[512];                       /* used by recvfrom() */
int cli_len = sizeof(cli);           /* used by recvfrom() */
int nbytes;                          /* used by recvfrom() */

/* 1) create the socket */
/* 2) bind to the socket */

nbytes = recvfrom(fd, buf, sizeof(buf), 0 /* flags */,
                  (struct sockaddr*) &cli, &cli_len);
if(nbytes < 0) {
      perror("recvfrom"); exit(1);
}
```

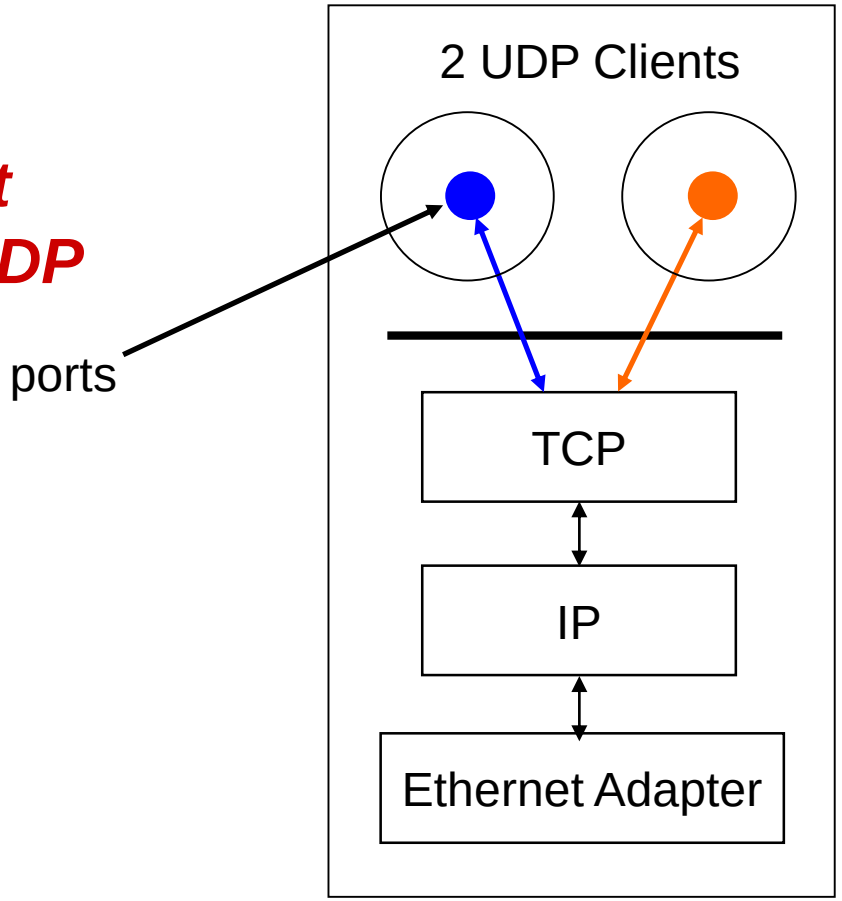
Socket I/O: recvfrom() continued...

```
nbytes = recvfrom(fd, buf, sizeof(buf), 0 /* flags */,
                  (struct sockaddr*) cli, &cli_len);
```

- The actions performed by ***recvfrom***
  - returns the number of bytes read (***nbytes***)
  - copies ***nbytes*** of data into ***buf***
  - returns the address of the client (***cli***)
  - returns the length of ***cli*** (***cli_len***)
  - don't worry about flags

UDP Client Example

- **How does a *UDP client* communicate with a *UDP server*?**

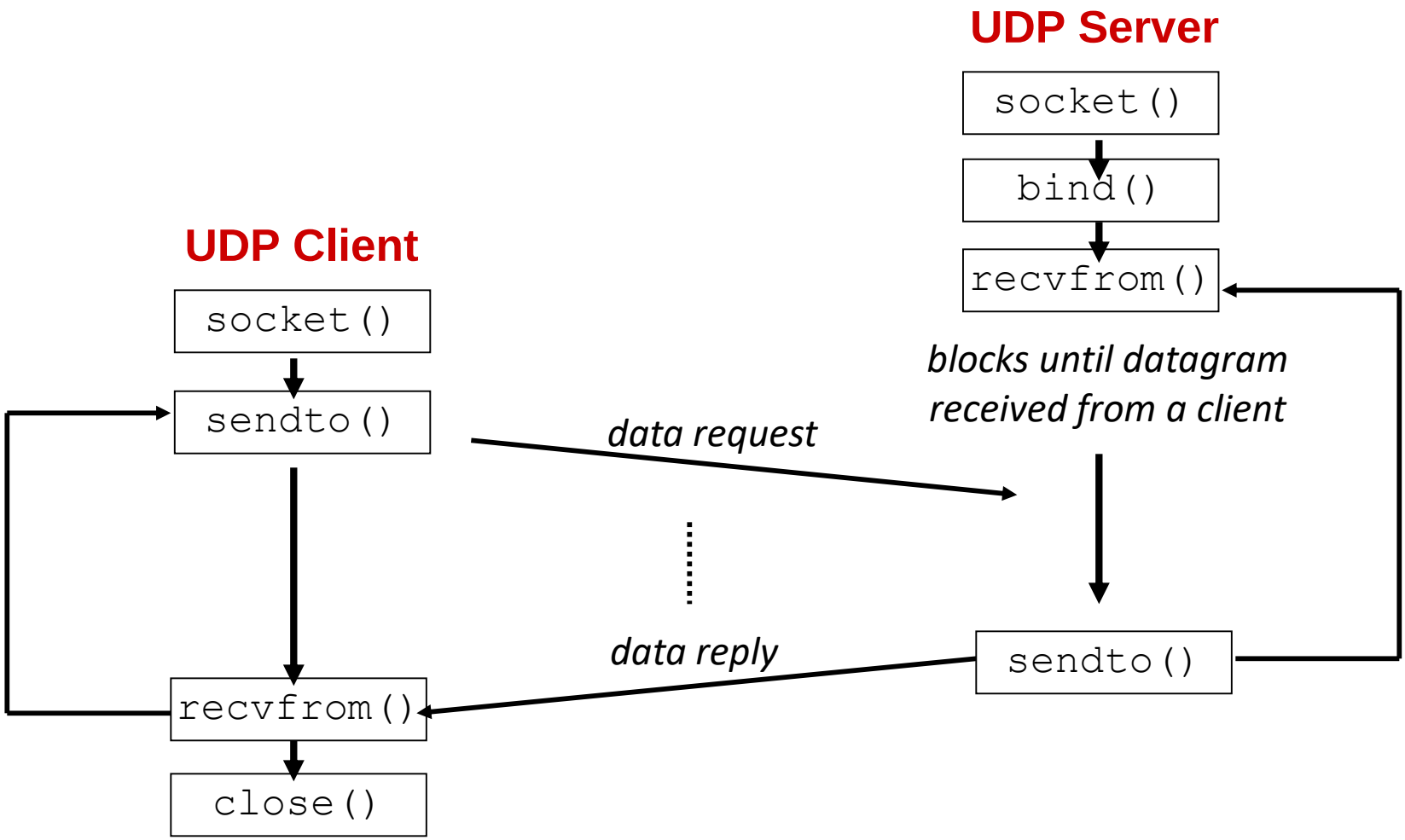


Socket I/O: sendto()

- ***write*** is not allowed
- Notice that the UDP client does not ***bind*** a port number
  - a port number is **dynamically assigned** when the first ***sendto*** is called

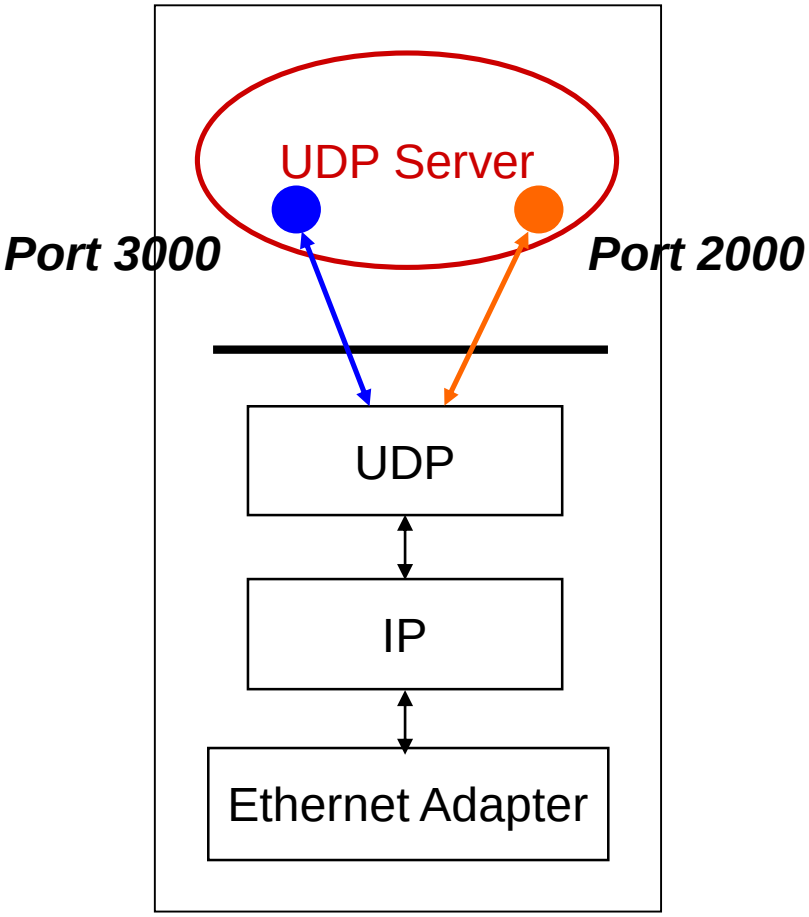
```
int fd;                                  /* socket descriptor */
struct sockaddr_in srv;                  /* used by sendto() */

/* 1) create the socket */

/* sendto: send data to IP Address "128.2.35.50" port 80 */
srv.sin_family = AF_INET;
srv.sin_port = htons(80);
srv.sin_addr.s_addr = inet_addr("128.2.35.50");

nbytes = sendto(fd, buf, sizeof(buf), 0 /* flags */,
                  (struct sockaddr*) &srv, sizeof(srv));
if(nbytes < 0) {
       perror("sendto");    exit(1);
}
```

Review: UDP Client-Server Interaction



from UNIX Network Programming Volume 1, figure 8.1

The UDP Server



- **How can the *UDP server* service multiple ports simultaneously?**

UDP Server: Servicing Two Ports

```
int s1;                              /* socket descriptor 1 */
int s2;                              /* socket descriptor 2 */

/* 1) create socket s1 */
/* 2) create socket s2 */
/* 3) bind s1 to port 2000 */
/* 4) bind s2 to port 3000 */

while(1) {
      recvfrom(s1, buf, sizeof(buf), ...);
      /* process buf */

      recvfrom(s2, buf, sizeof(buf), ...);
      /* process buf */
}
```

- **What problems does this code have?**

Socket I/O: select()

```
int select(int maxfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);

FD_CLR(int fd, fd_set *fds);   /* clear the bit for fd in fds */
FD_ISSET(int fd, fd_set *fds); /* is the bit for fd in fds? */
FD_SET(int fd, fd_set *fds);   /* turn on the bit for fd in fds */
FD_ZERO(fd_set *fds);          /* clear all bits in fds */
```

- ***maxfds***: number of descriptors to be tested
  - descriptors (0, 1, ... maxfds-1) will be tested
- ***readfds***: a set of *fds* we want to check if data is available
  - returns a set of *fds* ready to read
  - if input argument is *NULL*, not interested in that condition
- ***writefds***: returns a set of *fds* ready to write
- ***exceptfds***: returns a set of *fds* with exception conditions

Socket I/O: select()

```
int select(int maxfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);

struct timeval {
      long tv_sec;       /* seconds /
      long tv_usec;            /* microseconds */
}
```

- ***timeout***
  - if NULL, wait forever and return only when one of the descriptors is ready for I/O
  - otherwise, wait up to a fixed amount of time specified by *timeout*
    - if we don't want to wait at all, create a timeout structure with timer value equal to 0

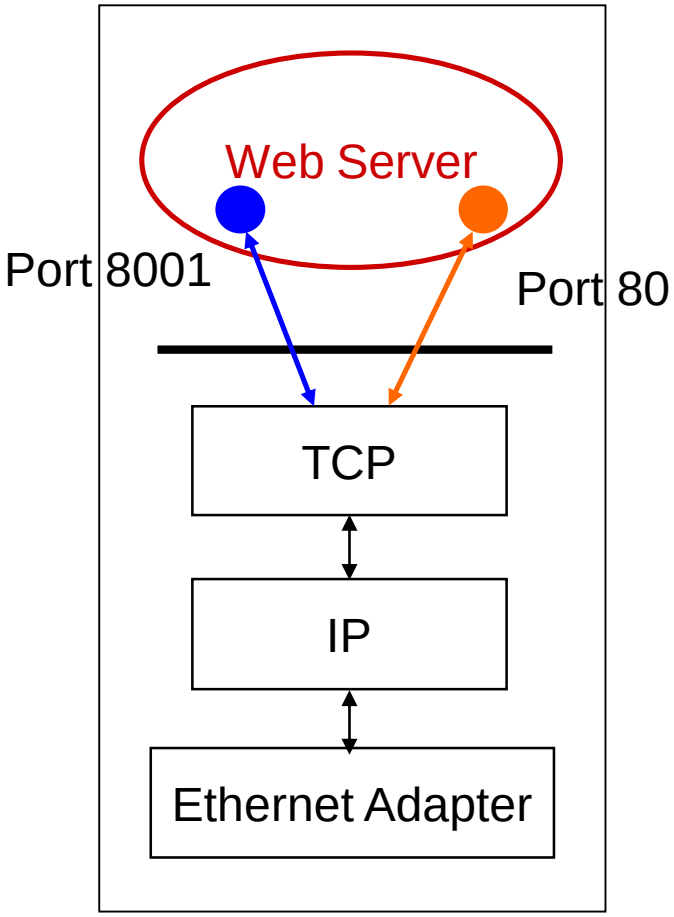
- Refer to the man page for more information

Socket I/O: select()

- ***select*** allows synchronous I/O multiplexing

```
int s1, s2;                          /* socket descriptors */
fd_set readfds;          /* used by select() */

/* create and bind s1 and s2 */
while(1) {
        FD_ZERO(&readfds);                /* initialize the fd set
*/
        FD_SET(s1, &readfds);   /* add s1 to the fd set */
        FD_SET(s2, &readfds);   /* add s2 to the fd set */

        if(select(s2+1, &readfds, 0, 0, 0) < 0) {
                perror("select");
                exit(1);
        }
        if(FD_ISSET(s1, &readfds)) {
                recvfrom(s1, buf, sizeof(buf), ...);
                /* process buf */
        }
        /* do the same for s2 */
}
```

More Details About a Web Server



How can a a *web server* manage multiple connections simultaneously?

Socket I/O: select()

```
int fd, next=0;                                  /* original socket */
int newfd[10];                                /* new socket descriptors */
while(1) {
        fd_set readfds;
        FD_ZERO(&readfds); FD_SET(fd, &readfds);

        /* Now use FD_SET to initialize other newfd's
           that have already been returned by accept() */

        select(maxfd+1, &readfds, 0, 0, 0);
        if(FD_ISSET(fd, &readfds)) {
                newfd[next++] = accept(fd, ...);
        }
        /* do the following for each descriptor newfd[n] */
        if(FD_ISSET(newfd[n], &readfds)) {
                read(newfd[n], buf, sizeof(buf));
                /* process data */
        }
}
```

- **Now the web server can support multiple connections…**

Remote Procedure Call

Message-oriented Protocols

- Many still in widespread use
  - Traditional TCP/IP and Internet protocols
- Difficult to design and implement
  - Especially with more sophisticated middleware
- Many difficult implementation issues for each new implementation
  - Formatting
  - Uniform representation of data
  - Client-server relationships
  - …

Remote Procedure Call (RPC)

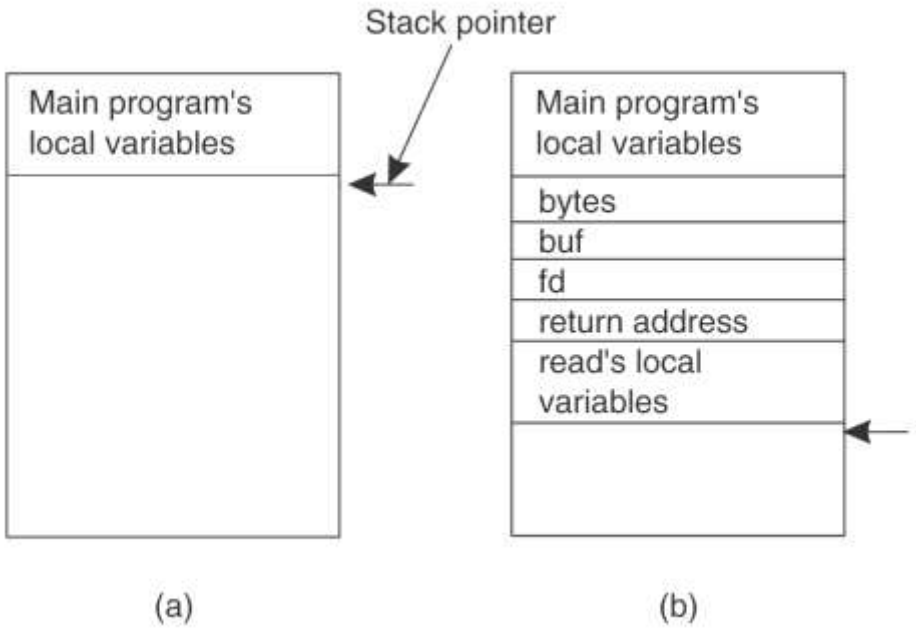
- *The* most common framework for newer protocols and for middleware
- Used both by operating systems and by applications
  - NFS is implemented as a set of RPCs
  - DCOM, CORBA, Java RMI, etc., are just RPC systems

Remote Procedure Call (RPC)

- Fundamental idea: –
  - Server process exports an *interface* of procedures or functions that can be called by client programs
    - similar to library API, class definitions, etc.
- Clients make local procedure/function calls
  - *As if* directly linked with the server process
  - Under the covers, procedure/function call is converted into a message exchange with remote server process

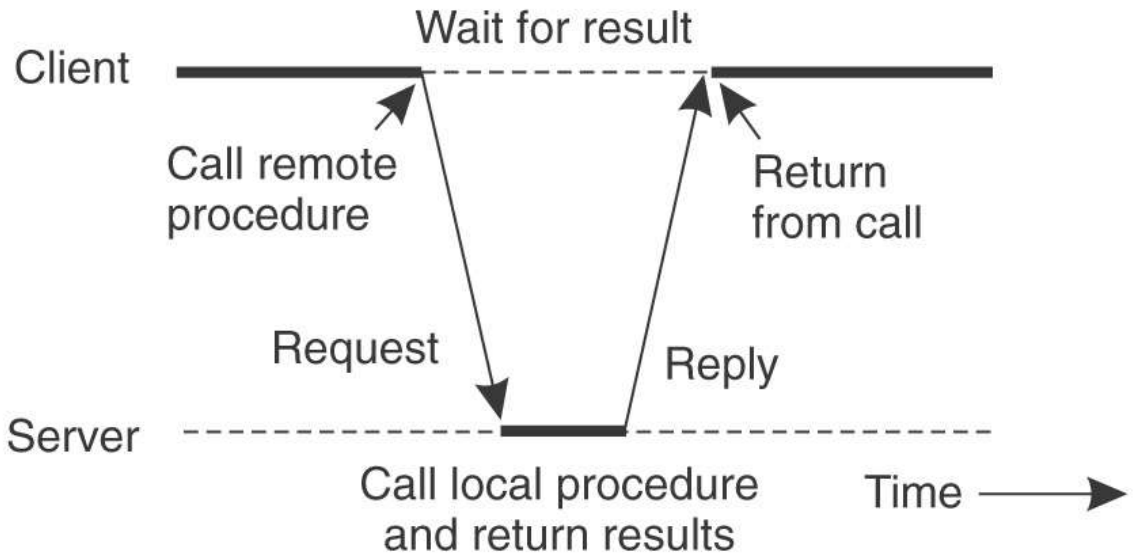
Ordinary procedure/function call

```
count = read(fd, buf, nbytes)
```



Remote Procedure Call

- Would like to do the same if called procedure or function is on a remote server



Solution — a pair of *Stubs*

- Client-side stub
  - Looks like local server function
  - Same interface as local function
  - Bundles arguments into message, sends to server-side stub
  - Waits for reply, un-bundles results
  - returns
- Server-side stub
  - Looks like local client function to server
  - Listens on a socket for message from client stub
  - Un-bundles arguments to local variables
  - Makes a local function call to server
  - Bundles result into reply message to client stub

Result

- The hard work of building messages, formatting, uniform representation, etc., is buried in the stubs
    - Where it can be automated!
- Client and server designers can concentrate on the semantics of application
- Programs behave in familiar way

RPC – Issues

- How to make the “remote” part of RPC invisible to the programmer?
- What are semantics of parameter passing?
  - E.g., pass by reference?
- How to bind (locate & connect) to servers?
- How to handle heterogeneity?
  - OS, language, architecture, …
- How to make it go fast?

RPC Model

- A server defines the service interface using an *interface definition language* (IDL)
  - the IDL specifies the names, parameters, and types for all client-callable server procedures
- A *stub compiler* reads the IDL declarations and produces two *stub functions* for each server function
  - *Server-side* and *client-side*

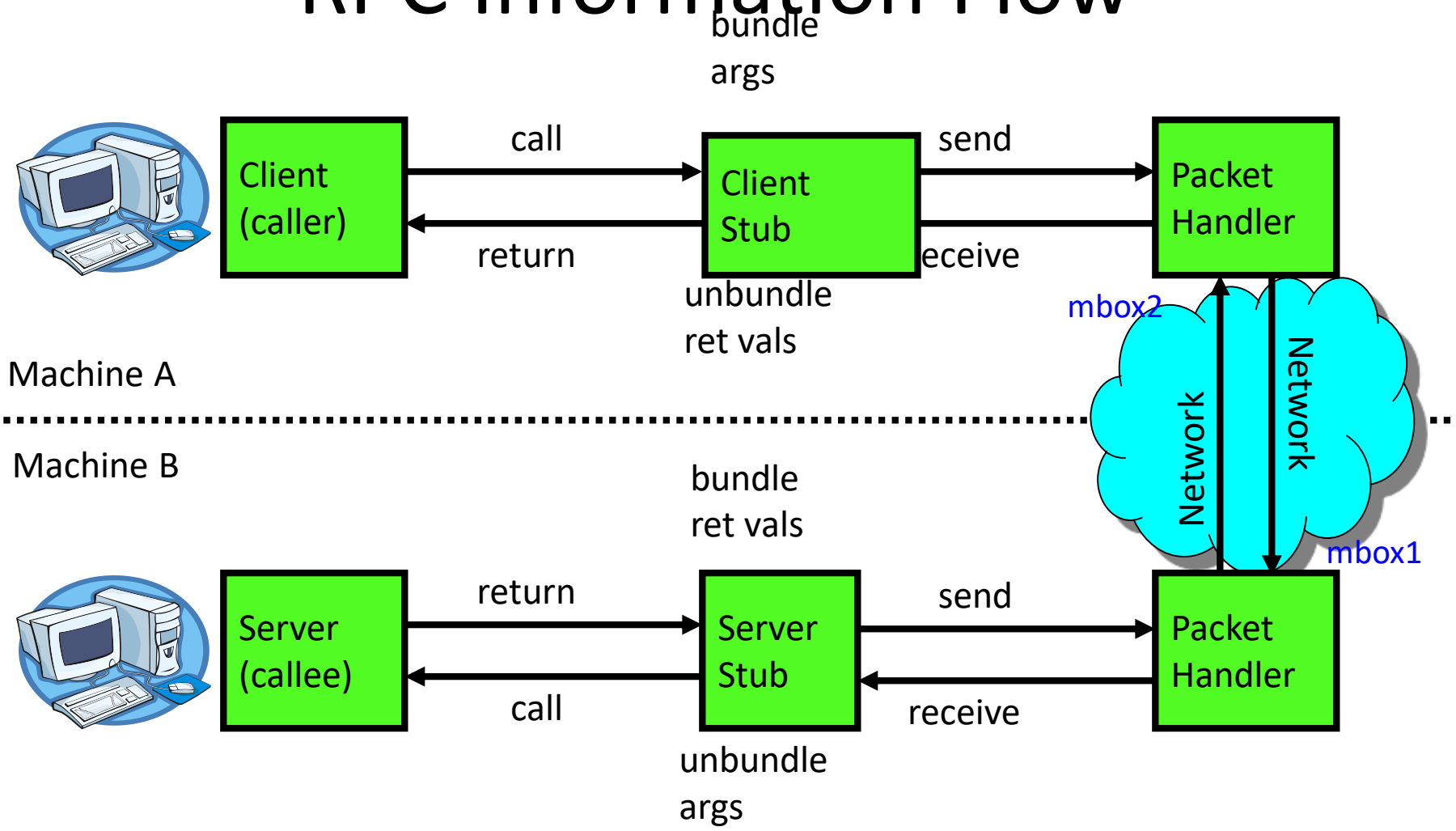
RPC Model (continued)

- Linking:–
  - Server programmer implements the service's functions and links with the *server-side* stubs
  - Client programmer implements the client program and links it with *client-side* stubs
- Operation:–
  - Stubs manage all of the details of remote communication between client and server

RPC Stubs

- A *client-side stub* is a function that looks to the client as if it were a callable server function
  - I.e., same API as the server's implementation of the function
- A *server-side stub* looks like a caller to the server
  - I.e., like a hunk of code invoking the server function
- The client program thinks it's invoking the server
  - but it's calling into the client-side stub
- The server program thinks it's called by the client
  - but it's really called by the server-side stub
- The stubs send messages to each other to make the RPC happen transparently (almost!)

RPC Information Flow

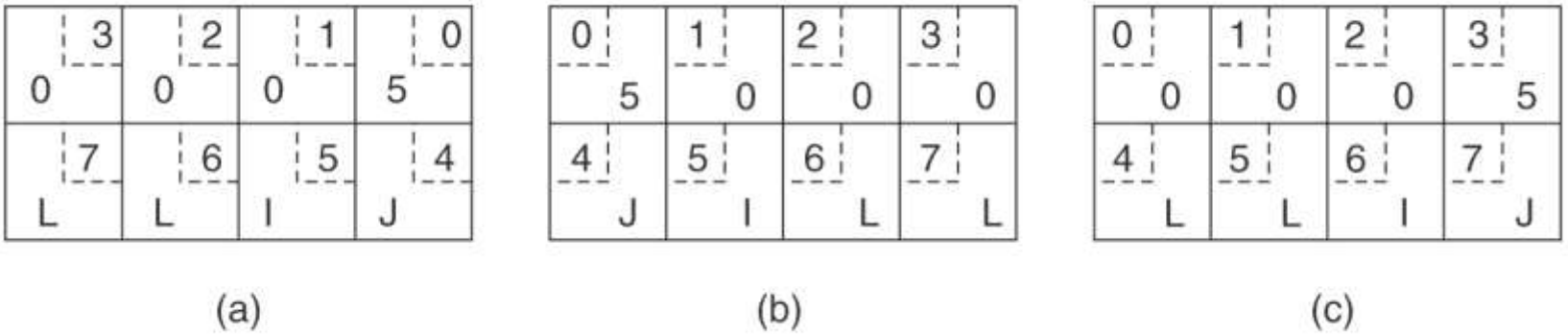


Marshalling Arguments

- *Marshalling* is the packing of function parameters into a message packet
  - the RPC stubs call type-specific functions to marshal or unmarshal the parameters of an RPC
    - Client stub marshals the arguments into a message
    - Server stub unmarshals the arguments and uses them to invoke the service function
  - on return:
    - the server stub marshals return values
    - the client stub unmarshals return values, and returns to the client program

Issue #1 — representation of data

- Big endian *vs.* little endian



Sent by Pentium

Rec'd by SPARC

After inversion

Representation of Data (continued)

- IDL must also define representation of data on network
  - Multi-byte integers
  - Strings, character codes
  - Floating point, complex, …
  - …
    - example: Sun's XDR (external data representation)
- Each stub converts machine representation to/from network representation
- Clients and servers must *not* try to cast data!

Issue #2 — Pointers and References

`read(int fd, char* buf, int nbytes)`

- Pointers are only valid within one address space
- Cannot be interpreted by another process
    - Even on same machine!

- Pointers and references are ubiquitous in C, C++
    - Even in Java implementations!

Pointers and References — Restricted Semantics

- Option: *call by value*
  - Sending stub dereferences pointer, copies result to message
  - Receiving stub conjures up a new pointer
- Option: *call by result*
  - Sending stub provides buffer, called function puts data into it
  - Receiving stub copies data to caller's buffer as specified by pointer

Pointers and References — Restricted Semantics (continued)

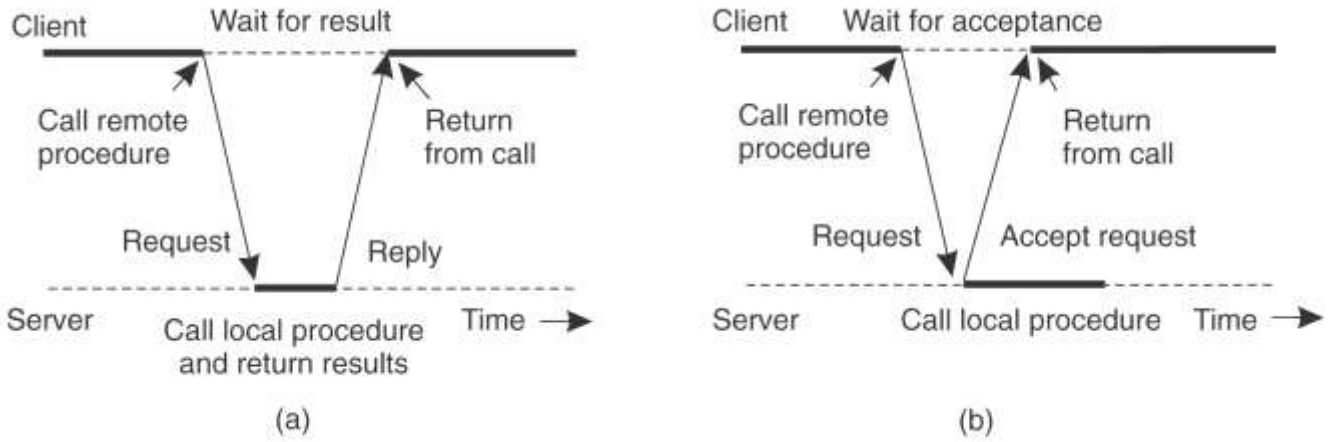
- Option: *call by value-result*
  - Caller's stub copies data to message, then copies result back to client buffer
  - Server stub keeps data in own buffer, server updates it; server sends data back in reply

- Not allowed:–
  - *Call by reference*

Transport of Remote Procedure Call

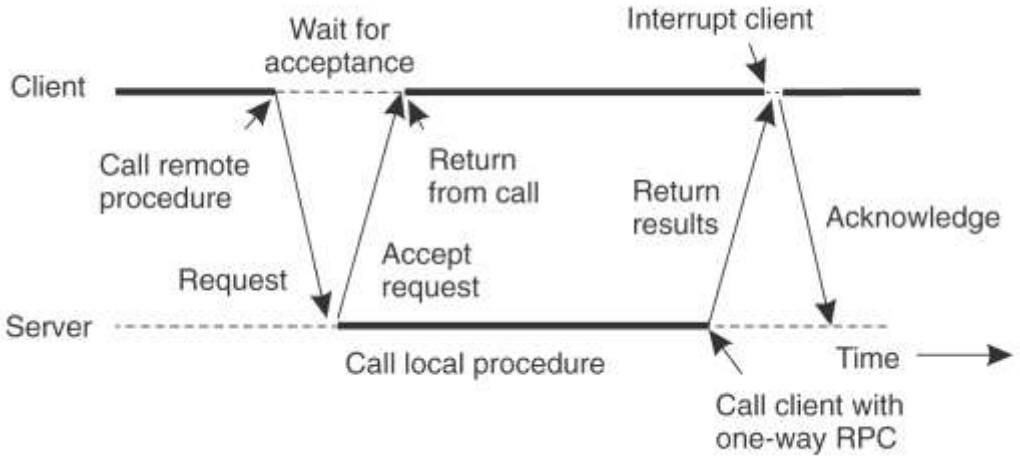
- Option — TCP
  - Connection-based, reliable transmission
  - Useful but heavyweight, less efficient
  - Necessary if repeating a call produces different result
- Alternative — UDP
  - If message fails to arrive within a reasonable time, caller's stub simply sends it again
  - Okay if repeating a call produces same result

Asynchronous RPC



- Analogous to spawning a thread
- Caller must eventually *wait* for result
  - Analogous to *join*

Asynchronous RPC (continued)



- Analogous to spawning a thread
- Caller must eventually *wait* for result
  - Analogous to *join*
  - Or be interrupted (software interrupt)

RPC *Binding*

- Binding is the process of connecting the client to the server
  - the server, when it starts up, exports its interface
    - identifies itself to a *network name server*
    - tells *RPC runtime* that it is alive and ready to accept calls
  - the client, before issuing any calls, imports the server
    - RPC runtime uses the name server to find the location of the server and establish a connection
- The import and export operations are explicit in the server and client programs

Remote Procedure Call is used …

- Between processes on different machines
  - E.g., client-server model
- Between processes on the same machine
  - More structured than simple message passing
- Between subsystems of an operating system
  - Windows XP (called *Local Procedure Call*)

POSIX Thread Programming

- Standard Thread Library for POSIX-compliant systems
- Supports thread creation and management
- Synchronization using
  - mutex variables
  - condition variables
- At the time of creation, different attributes can be assigned to
  - threads
  - mutex/condition variables

Using Posix Thread Library

- To use this library, `#include <pthread.h>` in your program.

- To compile, link with the pthread library:

```
gcc hello.c -o hello -lpthread
```

Data Types in POSIX

- special data type for threads (`pthread_t`)
- mutex variables for mutual exclusion (`pthread_mutex_t`)
  - mutex variables are like *binary semaphores*
  - a mutex variable can be in either *locked* or *unlocked* state
- condition variables using which thread can sleep until some other thread signals the condition (`pthread_cond_t`)
- various kind of attribute types used when initializing:
  - threads (`pthread_attr_t`)
  - mutex variables (`pthread_mutexattr_t`)
  - condition variables (`pthread_condattr_t`)

Functions and Data Types

- All POSIX thread functions have the form:

  pthread[_object] _operation

- Most of the POSIX thread library functions return 0 in case of success and some non-zero error-number in case of a failure.

Threads and Their Attributes

- *pthread_create()* function is used to create a new thread.
- A thread is created with specification of certain attributes such as:
  - Detach state (default non-detached)
  - Stack address
  - Stack size

```
int pthread_create (pthread_t *thread_id,
                    const pthread_attr_t *attributes,
                    void *(*thread_function)(void *),
                    void *arguments);
```

Example

```
#include <stdio.h>
#include <pthread.h>
main()  {
  pthread_t f2_thread, f1_thread, f3_thread; int i1=1,i2=2;
  void *f2(), *f1(),*f3();
  pthread_create(&f1_thread,NULL,f1,&i1);
  pthread_create(&f2_thread,NULL,f2,&i2);
  pthread_create(&f3_thread,NULL,f3,NULL);
  …
}
void *f1(int *i){
…
}
void *f2(int *i){
…
}

void *f3() {
}
```

Joining and Exiting

- A thread can wait for the completion of a non-detached thread by using

  ```
  pthread_join ( pthread_t thread, void **status)
  ```

  (All threads are created non-detached by default, so they are “joinable” by default).
- If any thread executes the system call `exit( ),` the entire process terminates.
- If the main thread completes its execution, it implicitly calls `exit( ),` and this again terminates the process.
- A thread (the main, or another thread) can exit by calling `pthread_exit( ),` this does not terminate the process.

Example

```
#include <stdio.h>
#include <pthread.h>
main()  {
  pthread_t f2_thread, f1_thread;
  void *f2(), *f1();
  pthread_create(&f1_thread,NULL,f1,NULL);
  pthread_create(&f2_thread,NULL,f2,NULL);
  pthread_join(f1_thread,NULL);
  pthread_join(f1_thread,NULL);
  pthread_exit(0);
}
void *f1(){
…
  pthread_exit(0);
}
void *f2(){
…
  pthread_exit(0);
}
```

Setting Thread Attributes

- Define and initialize attribute object:

```
pthread_attr_t  attr;
pthread_attr_init (&attr );
```

- For example, set the detach state:

```
pthread_attr_setdetachstate(&attr,
   THREAD_CREATE_DETACHED );
```

- Or, you can use “default attributes” when creating the thread.

Mutex Variables

- Used for mutual exclusion locks.
- A mutex variable can be either *locked* or *unlocked*

```
pthread_mutex_t lock;
```
// lock is a mutex variable

- Lock operation

```
pthread_mutex_lock( &lock ) ;
```

- Unlock operation

```
pthread_mutex_unlock( &lock )
```

- Initialization of a mutex variable by default attributes

```
pthread_mutex_init( &lock, NULL );
```

Example

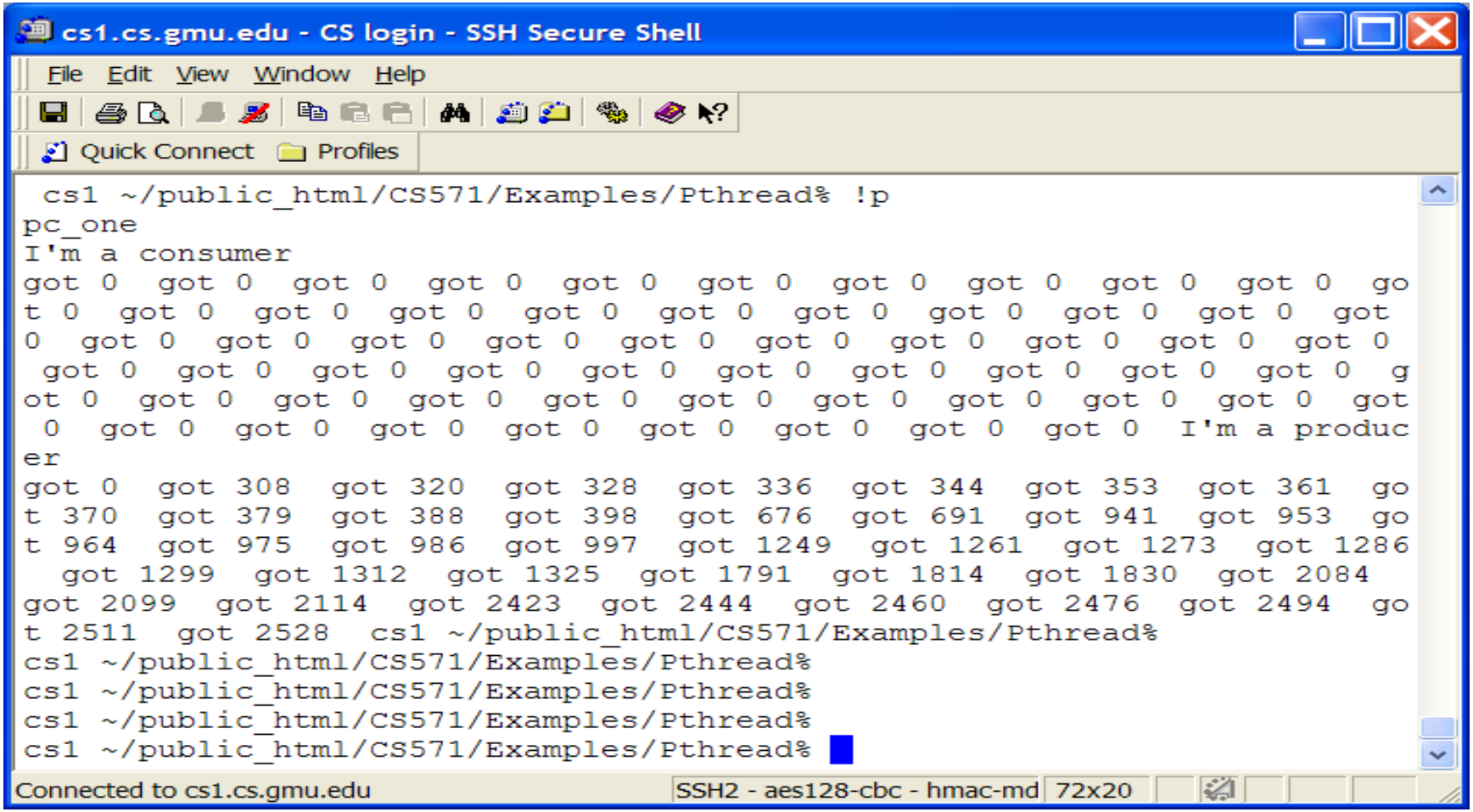
```
#include <stdio.h>
#include <pthread.h>
pthread_mutex_t region_mutex = PTHREAD_MUTEX_INITIALIZER;
int b;  /* buffer size = 1; */
 main()  {
  pthread_t producer_thread, consumer_thread;
  void *producer(),
  void *consumer();
  pthread_create(&consumer_thread,NULL,consumer,NULL);
  pthread_create(&producer_thread,NULL,producer,NULL);
  pthread_join(consumer_thread,NULL);
}
void put_buffer(int i){
  b = i;
}
int get_buffer(){
  return b ;
}
```

Example

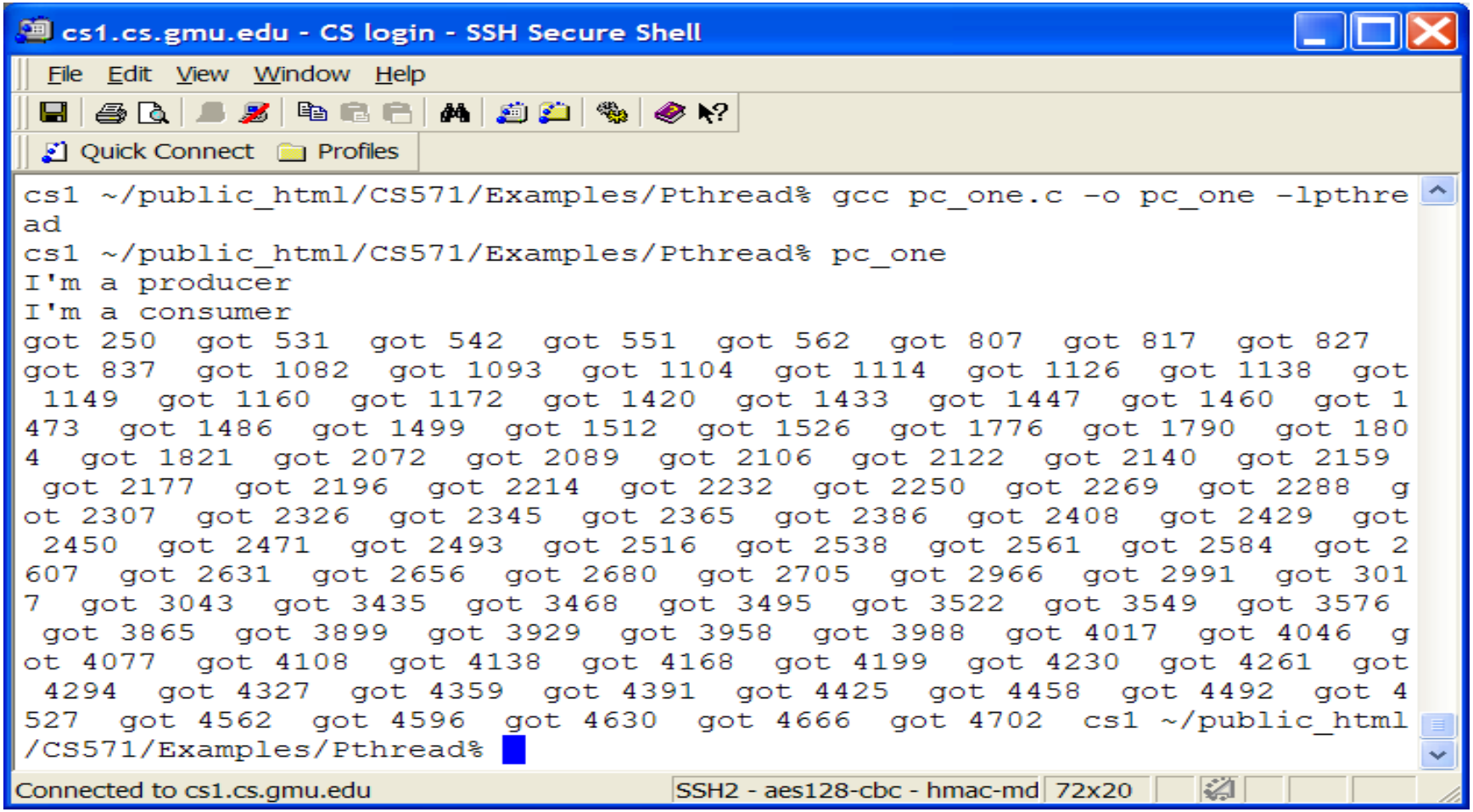
```
void *producer(){
int i = 0;
while (1) {
  pthread_mutex_lock(&region_mutex);
  put_buffer(i);
  pthread_mutex_unlock(&region_mutex);
  i++;
  }
}
void *consumer(){
int i,v;
for (i=0;i<100;i++) {
  pthread_mutex_lock(&region_mutex);
  v = get_buffer();
  pthread_mutex_unlock(&region_mutex);
  printf("got %d   ",v);
  }
}
```

Competition synchronization

Example output



Example output



Reader/Writer

```
pthread_mutex_t rw_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t reader_mutex = PTHREAD_MUTEX_INITIALIZER;
int num_readers = 0;
main()
{ …}

void *reader()
{  while  (1) {
    pthread_mutex_lock(&reader_mutex);
    num_readers++;
    if (num_readers == 1) pthread_mutex_lock(&rw_mutex);
    pthread_mutex_unlock(&reader_mutex);
    /* read */
    pthread_mutex_lock(&reader_mutex);
    num_readers--;
    if (num_readers == 0) pthread_mutex_unlock(&rw_mutex);
    pthread_mutex_unlock(&reader_mutex);
  }
}
void *writer()
{  while ( 1) {
    pthread_mutex_lock(&rw_mutex);
    /* write */
    pthread_mutex_unlock(&rw_mutex);
  }
}
```

Condition Variables

- In a critical section (i.e. where a mutex has been used), a thread can suspend itself on a *condition variable* if the state of the computation is not right for it to proceed.
  - It will suspend by *waiting* on a condition variable.
  - It will, however, release the critical section lock (mutex) .
  - When that condition variable is *signaled,* it will become ready again; it will attempt to reacquire that critical section lock and only then will be able proceed.

- With Posix threads, a condition variable can be associated with only one mutex variable!

Condition Variables

- `pthread_cond_t  SpaceAvailable;`
- `pthread_cond_init (&SpaceAvailable, NULL );`

- `pthread_cond_wait (&condition, &mutex)`
- `pthread_cond_signal(&condition)`

  unblock one waiting thread on that condition variable (that thread should still get the “lock” before proceeding)

- `pthread_cond_broadcast(&condition)`

  unblock all waiting threads on that condition variable (now all of them will compete to get the “lock”)

Condition Variables

Example:

pthread_mutex_lock (&mutex);

. . . . .

pthread_cond_wait (&SpaceAvailable, &mutex);

// now proceed again

. . .

pthread_mutex_unlock(&mutex);

- Some other thread will execute:

pthread_cond_signal (&SpaceAvailable);

- The signaling thread has priority over any thread that may be awakened
  - – “Signal-and-continue” semantics

Producer-Consumer Problem

- Producer will produce a sequence of integers, and deposit each integer in a bounded buffer (implemented as an array).
- All integers are positive, 0..999.
- Producer will deposit -1 when finished, and then terminate.
- Buffer is of finite size: 5 in this example.
- Consumer will remove integers, one at a time, and print them.
- It will terminate when it receives -1.

Definitions and Globals

```
#include <sys/time.h>
#include <stdio.h>
#include <pthread.h>
#include <errno.h>
#define SIZE 10

pthread_mutex_t region_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t space_available = PTHREAD_COND_INITIALIZER;
pthread_cond_t data_available = PTHREAD_COND_INITIALIZER;

int b[SIZE];    /* buffer */
int size = 0;  /* number of full elements */
int front,rear=0;  /* queue */
```

Producer Thread

```
void *producer()
{
int i = 0;
while (1) {
  pthread_mutex_lock(&region_mutex);
  while (size == SIZE) {
     pthread_cond_broadcast(&data_available);
     pthread_cond_wait(&space_available,&region_mutex);
  }
  if(i>99) i=-1;
   add_buffer(i);

     pthread_cond_broadcast(&data_available);
  pthread_mutex_unlock(&region_mutex);
  if (i==-1) break;
  i = i + 1;
}
pthread_exit(NULL);
}
```

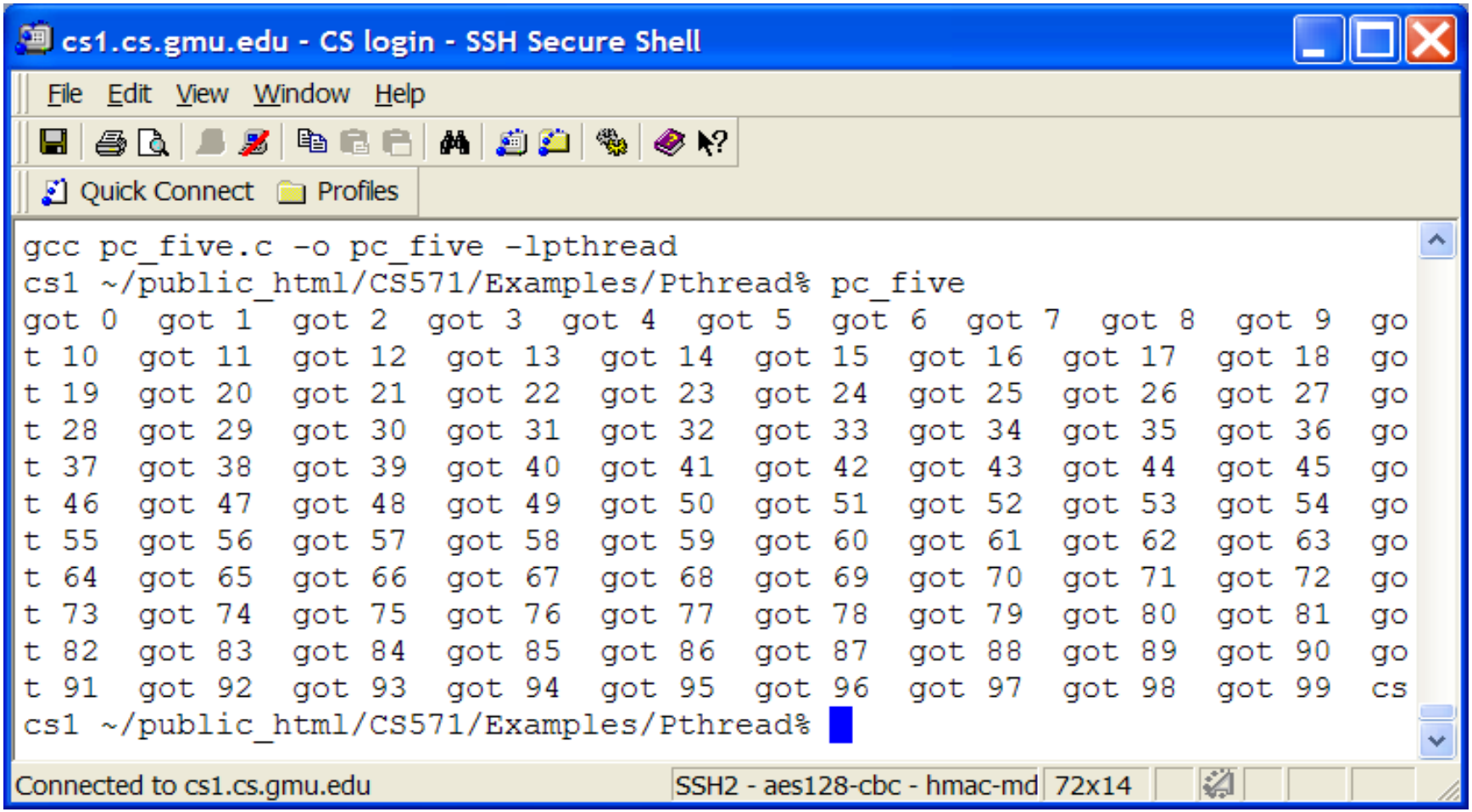
Consumer Thread

```
void *consumer()
{
int i,v;
While(1){
   pthread_mutex_lock(&region_mutex);
   while (size == 0) {
       pthread_cond_broadcast(&space_available);
       pthread_cond_wait(&data_available,&region_mutex);
   }
   v = get_buffer();
   pthread_cond_broadcast(&space_available);
   pthread_mutex_unlock(&region_mutex);
 if (v==-1) break;
   printf("got %d  ",v);
}
pthread_exit(NULL);
}
```

Main program

```
main()
{
  pthread_t producer_thread,consumer_thread;
  void *producer(),*consumer();
  pthread_create(&consumer_thread,NULL,consumer,NULL);
  pthread_create(&producer_thread,NULL,producer,NULL);
  pthread_join(consumer_thread,NULL);
}
void add_buffer(int i){
  b[rear] = i; size++;
  rear = (rear+1) % SIZE;
}
int get_buffer(){
  int v;
  v = b[front]; size--;
  front= (front+1) % SIZE;
  return v ;
}
```

Output


cs1.cs.gmu.edu - CS login - SSH Secure Shell
File Edit View Window Help
Quick Connect Profiles
gcc pc_five.c -o pc_five -lpthread
cs1 ~/public_html/CS571/Examples/Pthread% pc_five
got 0 got 1 got 2 got 3 got 4 got 5 got 6 got 7 got 8 got 9 go
t 10 got 11 got 12 got 13 got 14 got 15 got 16 got 17 got 18 go
t 19 got 20 got 21 got 22 got 23 got 24 got 25 got 26 got 27 go
t 28 got 29 got 30 got 31 got 32 got 33 got 34 got 35 got 36 go
t 37 got 38 got 39 got 40 got 41 got 42 got 43 got 44 got 45 go
t 46 got 47 got 48 got 49 got 50 got 51 got 52 got 53 got 54 go
t 55 got 56 got 57 got 58 got 59 got 60 got 61 got 62 got 63 go
t 64 got 65 got 66 got 67 got 68 got 69 got 70 got 71 got 72 go
t 73 got 74 got 75 got 76 got 77 got 78 got 79 got 80 got 81 go
t 82 got 83 got 84 got 85 got 86 got 87 got 88 got 89 got 90 go
t 91 got 92 got 93 got 94 got 95 got 96 got 97 got 98 got 99 cs
cs1 ~/public_html/CS571/Examples/Pthread%
Connected to cs1.cs.gmu.edu
SSH2 - aes128-cbc - hmac-md
72x14

Win32 vs. POSIX Interface

1. Function calls

Win32 Threads	PThreads
Just one type: HANDLE	Each object has its own data type : **pthread_t**, pthread_mutex_t, ...
One function needed to make one functionality (e.g. **WaitForSingleObject)**	Each object has its own functions/attributes
Simpler and more generic - is it OOP? (sounds like misusing **void*** in C).	Reading and understanding may be more straightforward and less confusing.

Win32 vs. POSIX Interface

2. Synchronization overhead

Win32 Threads	PThreads
Synchronization objects: – Events – Semaphores – Mutexes – Critical sections	**Synchronization primitives:** – Semaphores – Mutexes – Conditional variables
once an event is in the signaled state, it stays signaled. However, It is up to the programmer to ensure the proper switching of Windows events from the signaled to unsignaled state. (When an object is signaled, you have to check what other objects in the awaken thread might be waiting for and remove it from those wait queues)	*Signals to condition variables are either "caught" by waiting thread(s) or discarded.* However, use of a well known coding structure at each access of a condition variable will ensure no signals are "lost" by threads that may not be waiting at the exact time of signaling

More issues to consider

- Historically, hardware vendors have implemented their own proprietary versions of threads.
- In Windows, the thread is the basic execution unit, and the process is a container that holds this thread.
  In Linux, the basic execution unit is the process
- Some may claim POSIX threads are a low-level API and Windows threads are a high-level API
- In Windows the thread scheduling code is implemented in the kernel.
- In Linux realm, there are several drafts of the POSIX threads standard. Differences between drafts exist! Especially many scheduling mechanisms exist (user and kernel-level threads).
- The current POSIX standard is defined only for the C language.

Mapping WIN32 to PTHREADS

1. Process Mapping

Win32	Linux	Classification
CreateProcess() CreateProcessAsUser()	fork() setuid() exec()	Mappable
TerminateProcess()	kill()	Mappable
SetThreadpriority() GetThreadPriority()	Setpriority() getPriority()	Mappable
GetCurrentProcessID()	getpid()	Mappable
Exitprocess()	exit()	Mappable
Waitforsingleobject() Waitformultipleobject() GetExitCodeProcess()	waitpid() !! Using semaphores Waitforsingleobject / Waitformultipleobject can be implemented	Context specific
GetEnvironmentVariable SetEnvironmentVariable	getenv() setenv()	Mappable

Mappable: Both the Windows and Linux constructs provide similar functionality.
Context: The decision to use a specific Linux construct(s) depends on the application context

!! More on WaitForMultipleObjects may be found under the following reference:
http://www.ibm.com/developerworks/linux/library/l-ipc2lin3.html

Mapping WIN32 to PTHREADS

2. Thread Mapping

Win32	Linux	Classification
CreateThread	pthread_create pthread_attr_init pthread_attr_setstacksize pthread_attr_destroy	Mappable
ThreadExit	pthread_exit	Mappable
WaitForSingleObject	pthread_join pthread_attr_setdetachstate pthread_detach	Mappable
SetPriorityClass SetThreadPriority	setpriority sched_setscheduler sched_setparam pthread_setschedparam pthread_setschedpolicy pthread_attr_setschedparam pthread_attr_setschedpolicy	Context Specific

Mappable: Both the Windows and Linux constructs provide similar functionality.
Context: The decision to use a specific Linux construct(s) depends on the application context

Mapping WIN32 to PTHREADS

3. Synchronization

Win32 Thread Level	Linux Thread Level	Linux Process Level
Mutex	Mutex - pthread library	System V semaphores
Critical section	Mutex - pthread library	N/A *Critical sections are used only within threads of the same process*
Semaphore	Conditional Variable with mutex – pthreads POSIX semaphores	System V Semaphores
Event	Conditional Variable with mutex – pthreads POSIX semaphores	System V Semaphores

!! Conditional Variables: While mutexes implement synchronization by controlling thread access to data, condition variables allow threads to synchronize based upon the actual value of data.
See more under: https://computing.llnl.gov/tutorials/pthreads

Mapping WIN32 to PTHREADS

Synchronization (semaphores)

Win32 Thread	Linux Thread	Linux Process	Classification
CreateSemaphore	sem_init	semget semctl	Context specific
OpenSemaphore	N/A	semget	Context specific
WaitForSingleObject	sem_wait sem_trywait	semop	Context specific
ReleaseSemaphore	sem_post	semop	Context specific
CloseHandle	sem_destroy	semctl	Context specific

Notice: Win32 semaphores are created within a thread, and propagated all over the system. POSIX semaphores are wither for inter-thread within a process, or inter-process entities.

!! **opensemaphore** function enables multiple processes to open handles of the same semaphore object. The function succeeds only if some process has already created the semaphore by using the **CreateSemaphore** function.

Mapping WIN32 to PTHREADS
Synchronization (events)

Win32 Thread	Linux Thread	Linux Process	Classification
CreateEvent OpenEvent	pthread_cond_init sem_init	semget semctl	context specific
SetEvent	pthread_cond_signal sem_post	semop	context specific
ResetEvent	N/A	N/A	context specific
WaitForSingleObject	pthread_cond_wait pthread_cond_timedwait sem_wait sem_trywait	semop	context specific
CloseHandle	pthread_cond_destroy sem_destroy	semctl	context specific

Notice (See conclusion in the next slide).

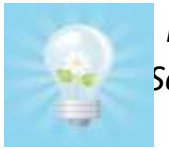
- Events are Windows specific objects.
- POSIX semaphores with count set to 1 provide similar functionality to the Windows events. However, they don't provide timeout in the wait functions.
- Conditional variables & Mutex in pthreads provide event-based synchronization between threads, but they are **synchronous**.

Mapping WIN32 to PTHREADS
Synchronization (events) - conclusion

Win32 Threads	PThreads
Both Manual/ Auto-reset events	Only Auto-reset mechanism.
Named events to synchronization processes. Un-named events to synchronize threads.	Synchronization is inter-thread based. (System V semaphore or signals can be used)
Event objects initial state is set to **signaled**.	Does not provide an initial state,. *(POSIX semaphores provide initial state)*
Event objects are **asynchronous**	Conditional variables are **synchronous** *(POSIX/System V events are asynchronous)*
Timeout value can be specified	Timeout value can be specified *(Other Linux system s don't support timeout)*

Mapping WIN32 to PTHREADS
Synchronization (mutex)

Win32 Thread	Linux Thread	Linux Process	Classification
CreateMutex	pthreads_mutex_init	semget semctl	context specific
OpenMutex	N/A	semget	context specific
WaitForSingleObject	pthread_mutex_lock pthread_mutex_trylock	semop	context specific
ReleaseMutex	pthread_mutex_unlock	semop	context specific
CloseHandle	pthread_mutex_destroy	semctl	context specific



Notice (See conclusion in the next slide).

Some major differences reside, including:

- Named and un-named mutexes.
- Ownershiop at creation.
- Timeout during wait.
- Recursive mutexes.

Mapping WIN32 to LINUX
Synchronization (mutex) - conclusion

Win32 Threads	PThreads
Named mutexes synchronize process/thread Un-named mutex synchronize threads.	Synchronization is inter-thread based. (System V semaphore or signals can be used)
Can be owned during creation.	To achieve the same in Linux, a mutex should be locked explicitly after creation.
Recursive by default	*Pthread has recursive mutex (initialized explicitly)* *(other Linux system do not allow recursive mutex)*
Timeout value can be specified	Timeout not available. *(can be obtained via application logic)*

Mapping WIN32 to LINUX
Synchronization (Critical Sections)

Win32 Thread	Linux Thread	Classification
InitializeCriticalSection InitializeCriticalSectionAndSpinCount	pthread_mutex_init	Mappable
EnterCriticalSection TryEnterCriticalSection	pthread_mutex_lock pthread_mutex_trylock	Mappable
LeaveCriticalSection	pthread_mutex_trylock	Mappable
DeleteCriticalSection	pthread_mutex_destroy	Mappable

Notice: Since the critical sections are used only between the threads of the same process, Pthreads mutex can be used to achieve the same **functionality** on Linux systems.

1 - In general, mutex objects are more CPU intensive and slower to use than critical sections due to a larger amount of bookkeeping, and the deep execution path into the kernel taken to acquire a mutex. The equivalent functions for accessing a critical section remain in thread context, and consist of merely checking and incrementing/decrementing lock-count related data. This makes critical sections light weight, fast, and easy to use for thread synchronization within a process.

2- In Win32 The primary benefit of using a mutex object is that a mutex can be named and shared across process boundaries.

References

- **POSIX Threads Programming**
  https://computing.llnl.gov/tutorials/pthreads/

- **Why pthreads are better than Win32**
  http://softwarecommunity.intel.com/ISN/Community/en-US/forums/post/840096.aspx

  Why Windows threads are better than pthreads:
  http://softwareblogs.intel.com/2006/10/19/why-windows-threads-are-better-than-posix-threads/

- **Port Windows IPC apps to Linux (including code examples)**

http://www.ibm.com/developerworks/linux/library/l-ipc2lin1.html
http://www.ibm.com/developerworks/linux/library/l-ipc2lin2.html
http://www.ibm.com/developerworks/linux/library/l-ipc2lin3.html

Introduction to OpenMP

Introduction to OpenMP

- What is OpenMP?
  - Open specification for Multi-Processing
  - “Standard” API for defining multi-threaded shared-memory programs
  - www.openmp.org – Talks, examples, forums, etc.

- High-level API
  - Preprocessor (compiler) directives (~ 80%)
  - Library Calls (~ 19%)
  - Environment Variables (~ 1%)

A Programmer's View of OpenMP

- OpenMP is a portable, threaded, shared-memory programming *specification* with "light" syntax
  - Exact behavior depends on OpenMP *implementation*!
  - Requires compiler support (C or Fortran)

- OpenMP will:
  - Allow a programmer to separate a program into *serial regions* and *parallel regions,* rather than T concurrently-executing threads.
  - Hide stack management
  - Provide synchronization constructs

- OpenMP will not:
  - Parallelize (or detect!) dependencies
  - Guarantee speedup
  - Provide freedom from data races

Outline

- Introduction
  - Motivating example
  - Parallel Programming is Hard
- OpenMP Programming Model
  - Easier than PThreads
- Microbenchmark Performance Comparison
  - vs. PThreads
- Discussion
  - specOMP

Current Parallel Programming

1. Start with a parallel algorithm
2. Implement, keeping in mind:
   - Data races
   - Synchronization
   - Threading Syntax
3. Test & Debug
4. Debug
5. Debug

Motivation – Threading Library

```
void* SayHello(void *foo) {
  printf( "Hello, world!\n" );
  return NULL;
}

int main() {
  pthread_attr_t attr;
  pthread_t threads[16];
  int tn;
  pthread_attr_init(&attr);
  pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
  for(tn=0; tn<16; tn++) {
    pthread_create(&threads[tn], &attr, SayHello, NULL);
  }
  for(tn=0; tn<16 ; tn++) {
    pthread_join(threads[tn], NULL);
  }
  return 0;
}
```

Motivation

- Thread libraries are hard to use
  - P-Threads/Solaris threads have many library calls for initialization, synchronization, thread creation, condition variables, etc.
  - Programmer must code with multiple threads in mind
- Synchronization between threads introduces a new dimension of program correctness

Motivation

- Wouldn't it be nice to write serial programs and somehow parallelize them "automatically"?
  - OpenMP can parallelize many serial programs with relatively few annotations that specify parallelism and independence
  - OpenMP is a small API that hides cumbersome threading calls with simpler *directives*

Better Parallel Programming

1. Start with *some* algorithm
   - Embarrassing parallelism is helpful, but not necessary
2. Implement serially, *ignoring*:
   - Data Races
   - Synchronization
   - Threading Syntax
3. Test and Debug
4. Automatically (*magically?*) parallelize
   - Expect linear speedup

Motivation – OpenMP

```
int main() {

  // Do this part in parallel

  printf( "Hello, World!\n" );

  return 0;
}
```

Motivation – OpenMP

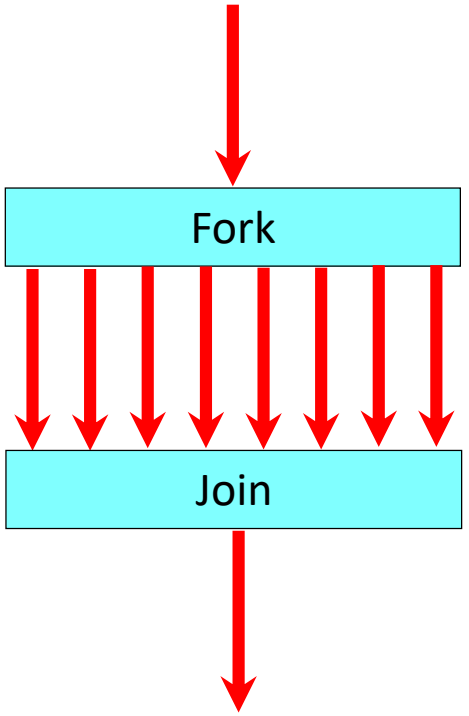
```
int main() {

  omp_set_num_threads(16);

  // Do this part in parallel
  #pragma omp parallel
  {
    printf( "Hello, World!\n" );
  }

  return 0;
}
```

OpenMP Parallel Programming

1. Start with *a parallelizable* algorithm
   - Embarrassing parallelism is good, loop-level parallelism is necessary
2. Implement serially, *mostly ignoring*:
   - Data Races
   - Synchronization
   - Threading Syntax
3. Test and Debug
4. Annotate the code with parallelization (and synchronization) directives
   - Hope for linear speedup
5. Test and Debug

Programming Model - Threading

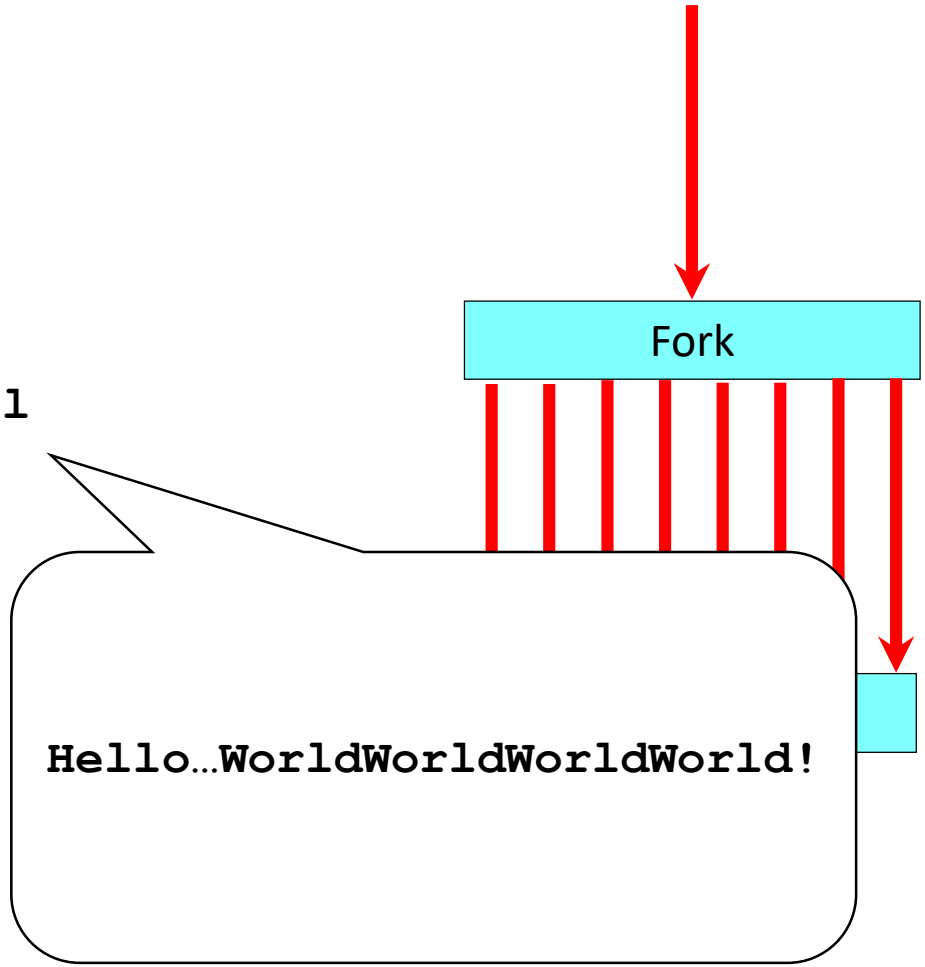
- Serial regions by default, annotate to create *parallel regions*
  - Generic parallel regions
  - Parallelized loops
  - Sectioned parallel regions



- Thread-like Fork/Join model
  - Arbitrary number of *logical* thread creation/ destruction events

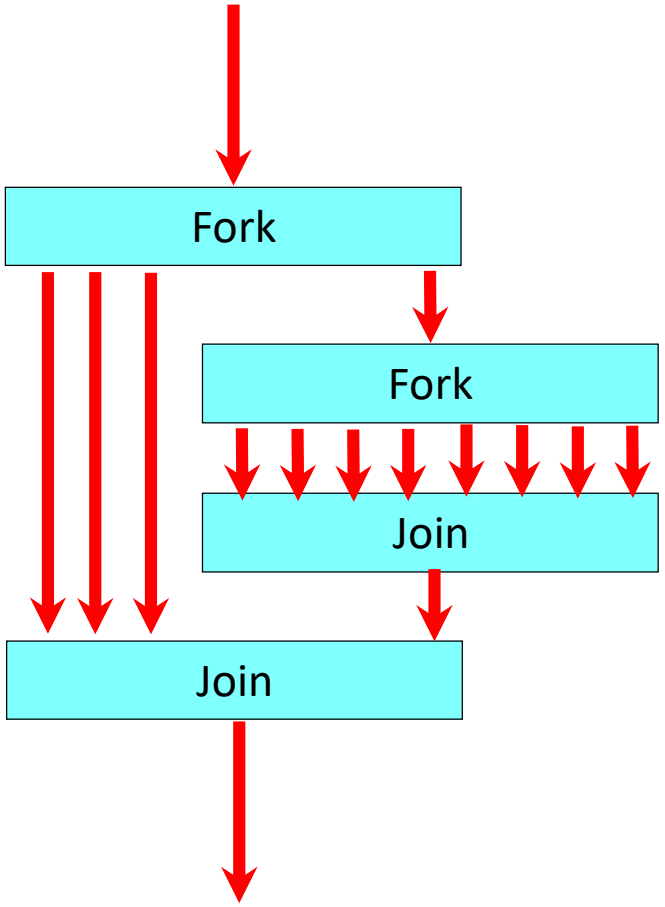
Programming Model - Threading

```
int main() {
  // serial region
  printf("Hello…");
  // parallel region
  #pragma omp parallel
  {
    printf("World");
  }
  // serial again
} printf("!");
```



Programming Model – Nested Threading

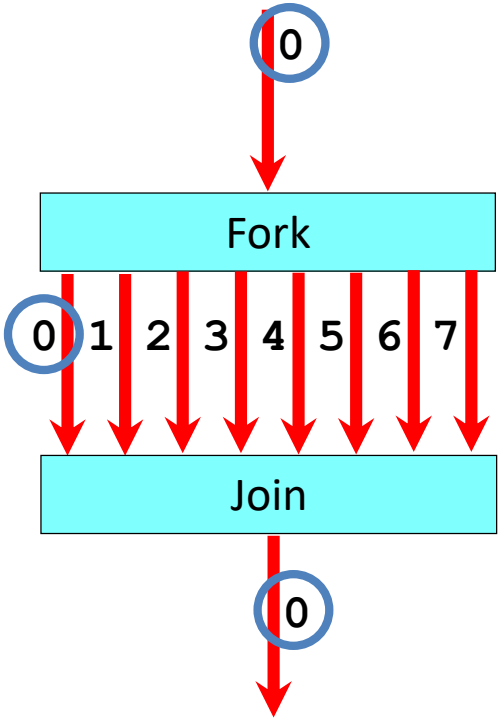
- Fork/Join can be nested
  - Nesting complication handled "automagically" at compile-time
  - Independent of the number of threads actually running



Programming Model – Thread Identification

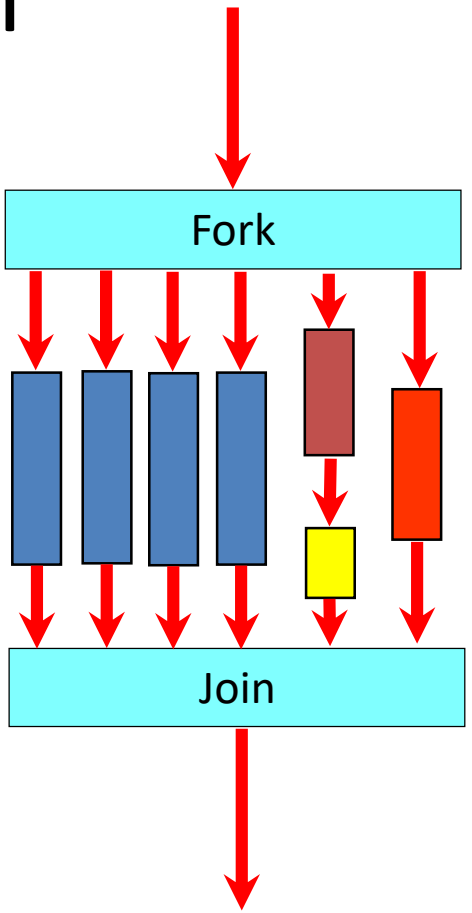
Master Thread

- Thread with ID=0
- Only thread that exists in sequential regions
- Depending on implementation, may have special purpose inside parallel regions
- Some special directives affect only the master thread (like `master`)



Programming Model – Data/Control Parallelism

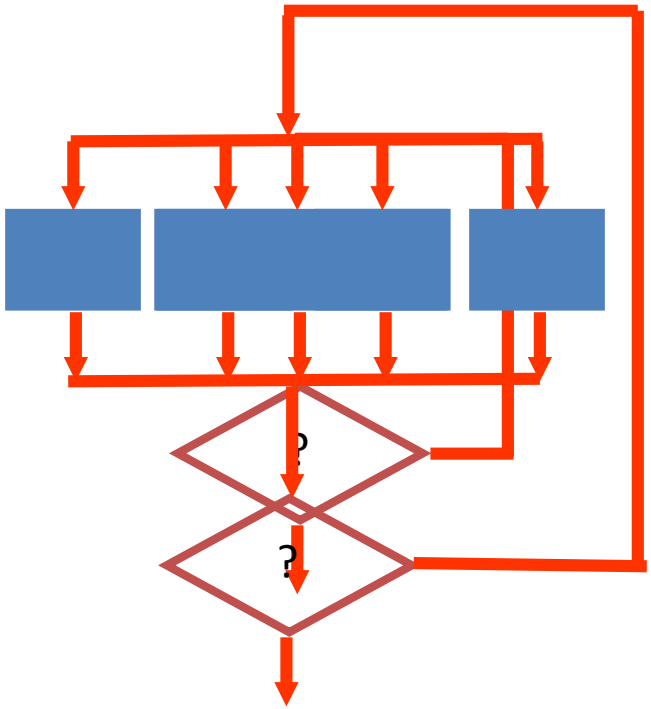
- Data parallelism
  - Threads perform similar functions, guided by thread identifier
- Control parallelism
  - Threads perform differing functions
    - One thread for I/O, one for computation, etc…



Programming Model – Concurrent Loops

- OpenMP easily parallelizes loops
  - No data dependencies between iterations!

- Preprocessor calculates loop bounds for each thread directly from *serial* source



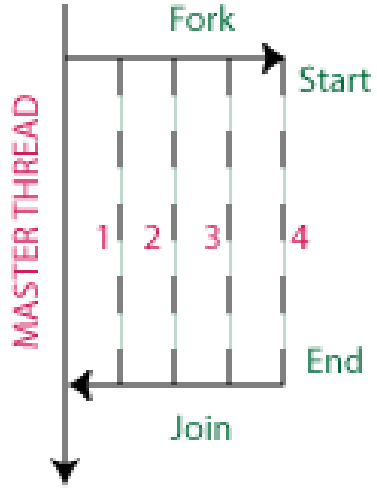
```
#pragma omp parallel for
for( i=0; i < 25; i++ ) {
  printf("Foo");
}
```

Sequential Execution
on a single thread


```
for (i=0; i<1000; i++)
      dest[i] = src1[i]*128 + src2[i];
```

End - Result

Execution
4 threads



Thread 1

```
for (i=0; i<250; i++)
      dest[i] = src1[i]*128 + src2[i];
```

Thread 2

```
for (j=250; j<500; j++)
      dest[j] = src1[j]*128 + src2[j];
```

Thread 3

```
for (k=500; k<750; k++)
      dest[k] = src1[k]*128 + src2[k];
```

Thread 4

```
for (l=750; l<1000; l++)
      dest[l] = src1[l]*128 + src2[l];
```

End - Result

Programming Model – Loop Scheduling

- `schedule` clause determines how loop iterations are divided among the thread team
  - `static([chunk])` divides iterations statically between threads
    - Each thread receives `[chunk]` iterations, rounding as necessary to account for all iterations
    - Default `[chunk]` is `ceil( # iterations / # threads )`
  - `dynamic([chunk])` allocates `[chunk]` iterations per thread, allocating an additional `[chunk]` iterations when a thread finishes
    - Forms a logical work queue, consisting of all loop iterations
    - Default `[chunk]` is 1
  - `guided([chunk])` allocates dynamically, but `[chunk]` is exponentially reduced with each allocation

Programming Model – Loop Scheduling

```
#pragma omp parallel for \
 schedule(static)
for( i=0; i<16; i++ )
{
  doIteration(i);
}
```

```
// Static Scheduling

int chunk = 16/T;
int base = tid * chunk;
int bound =
   (tid+1)*chunk;

for( i=base; i<bound; i++
   )
{
  doIteration(i);
}

Barrier();
```

Programming Model – Loop Scheduling

```
#pragma omp parallel for \
 schedule(dynamic)

 for( i=0; i<16; i++ )
 {
   doIteration(i);
 }
```

```
// Dynamic Scheduling

int current_i;

while( workLeftToDo() )
{
  current_i = getNextIter();
  doIteration(i);
}

Barrier();
```

Programming Model – Data Sharing

- Parallel programs often employ two types of data
  - Shared data, visible to all threads, similarly named
  - Private data, visible to a single thread (often stack-allocated)
- **PThreads:**
  - **Global-scoped variables are shared**
  - **Stack-allocated variables are private**

- **OpenMP:**
  - **`shared` variables are shared**
  - **`private` variables are private**

```
//int bigdata[1024];

int bigdata[1024];

void* foo(void* bar) {

void* foo(void* bar) {

  // private, stack

  int tid;#pragma omp parallel \

   shared ( bigdata ) \

  /* Calculation goes private ( tid )

  {  here */

}   /* Calc. here */

  }

}
```

Programming Model - Synchronization

- OpenMP Synchronization
  - OpenMP Critical Sections
    - Named or unnamed
    - No *explicit* locks
  - Barrier directives
  - Explicit Lock functions
    - When all else fails – may require `flush` directive
  - Single-thread regions *within* parallel regions
    - `master`, `single` directives

```
#pragma omp critical
{
  /* Critical code here */
}
#pragma omp barrier

omp_set_lock( lock l );
/* Code goes here */
omp_unset_lock( lock l );
#pragma omp single
{
  /* Only executed once */
}
```

Programming Model - Summary

- Threaded, shared-memory execution model
  - Serial regions and parallel regions
  - Parallelized loops with customizable scheduling
- Concurrency expressed with preprocessor directives
  - Thread creation, destruction mostly hidden
  - Often expressed *after writing* a serial version through annotation

Outline

- Introduction
  - Motivating example
  - Parallel Programming is Hard
- OpenMP Programming Model
  - Easier than PThreads
- Microbenchmark Performance Comparison
  - vs. PThreads
- Discussion
  - specOMP

Performance Concerns

- Is the overhead of OpenMP too high?
  - How do the scheduling and synchronization options affect performance?
  - How does autogenerated code compare to hand-written code?
- Can OpenMP scale?
  - 4 threads? 16? More?
- What should OpenMP be compared against?
  - PThreads?
  - MPI?

Performance Comparison: OMP vs. Pthreads

- PThreads
  - Shared-memory, portable threading implementation
  - Explicit thread creation, destruction (`pthread_create`)
  - Explicit stack management
  - Synch: Locks, Condition variables

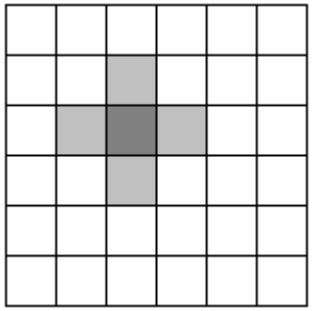
- Microbenchmarks implemented in OpenMP, PThreads
  - Explore OpenMP loop scheduling policies
  - Comparison vs. tuned PThreads implementation

Methodology

- Microbenchmarks implemented in OpenMP and PThreads, compiled with similar optimizations, same compiler (Sun Studio)
- Execution times measured on a 16-processor Sun Enterprise 6000 (`cabernet.cs.wisc.edu`), 2GB RAM, 1MB L2 Cache
- Parameters varied:
  - Number of processors (threads)
  - Working set size
  - OpenMP loop scheduling policy

Microbenchmark: Ocean

- Conceptually similar to SPLASH-2's ocean
- Simulates ocean temperature gradients via su approximation
  - Operates on a 2D grid of floating point values



- "Embarrassingly" Parallel
  - Each thread operates in a rectangular region
  - Inter-thread communication occurs only on region boundaries
  - Very little synchronization (barrier-only)
- Easy to write in OpenMP!

Microbenchmark: Ocean

```
  #pragma omp parallel for \
   shared(ocean,x_dim,y_dim) private(x,y)
for( t=0; t < t_steps; t++) {

  for( x=0; x < x_dim; x++) {
    for( y=0; y < y_dim; y++) {
      ocean[x][y] = /* avg of neighbors */
    }
  } // Implicit Barrier Synchronization
  temp_ocean = ocean;
  ocean = other_ocean;
  other_ocean = temp_ocean;
}
```

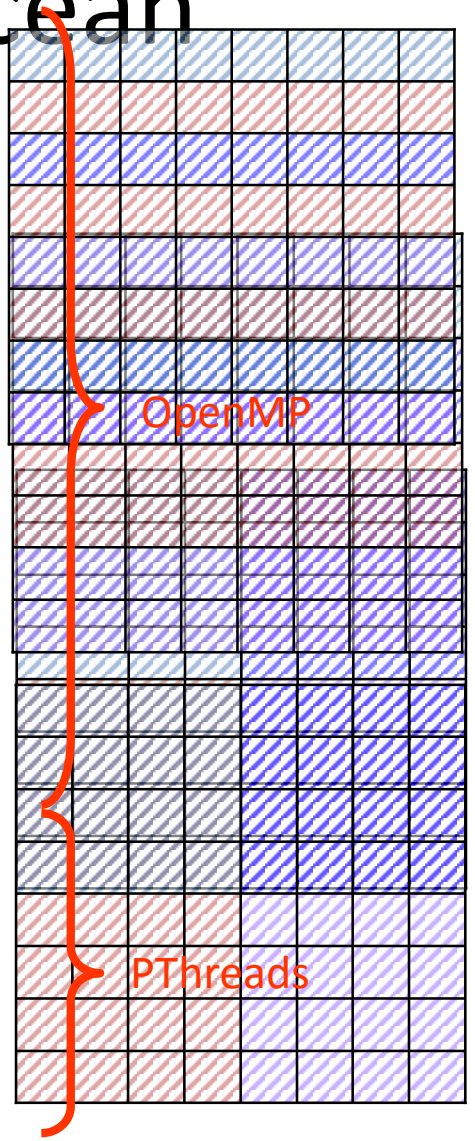
Microbenchmark: Ocean

- `ocean_dynamic` – Traverses entire ocean, row-by-row, assigning row iterations to threads with `dynamic` scheduling.

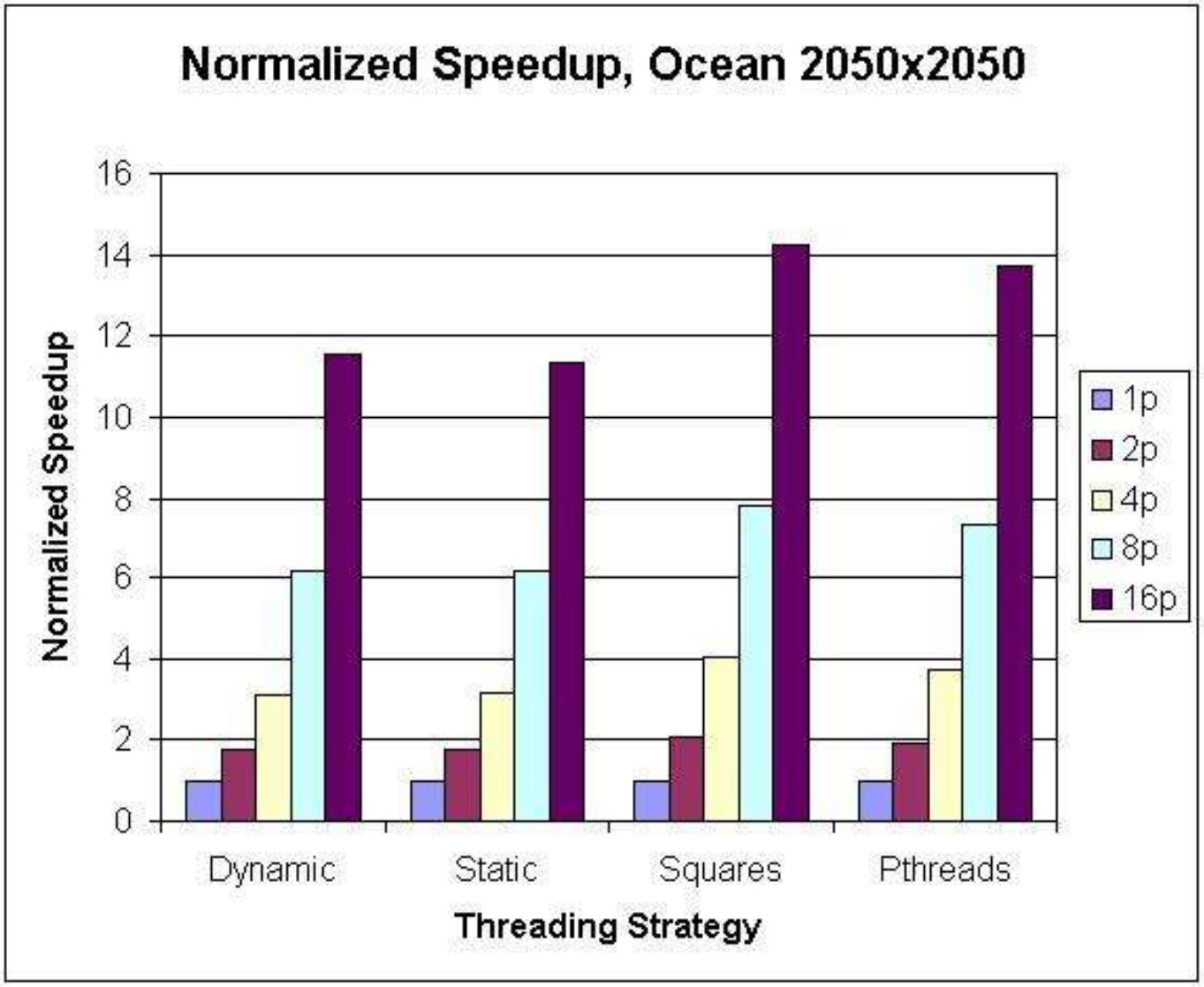
- **`ocean_static` – Traverses entire ocean, row-by-row, assigning row iterations to threads with `static` scheduling.**

- **`ocean_squares` – Each thread traverses a square-shaped section of the ocean. Loop-level scheduling not used—loop bounds for each thread are determined explicitly.**

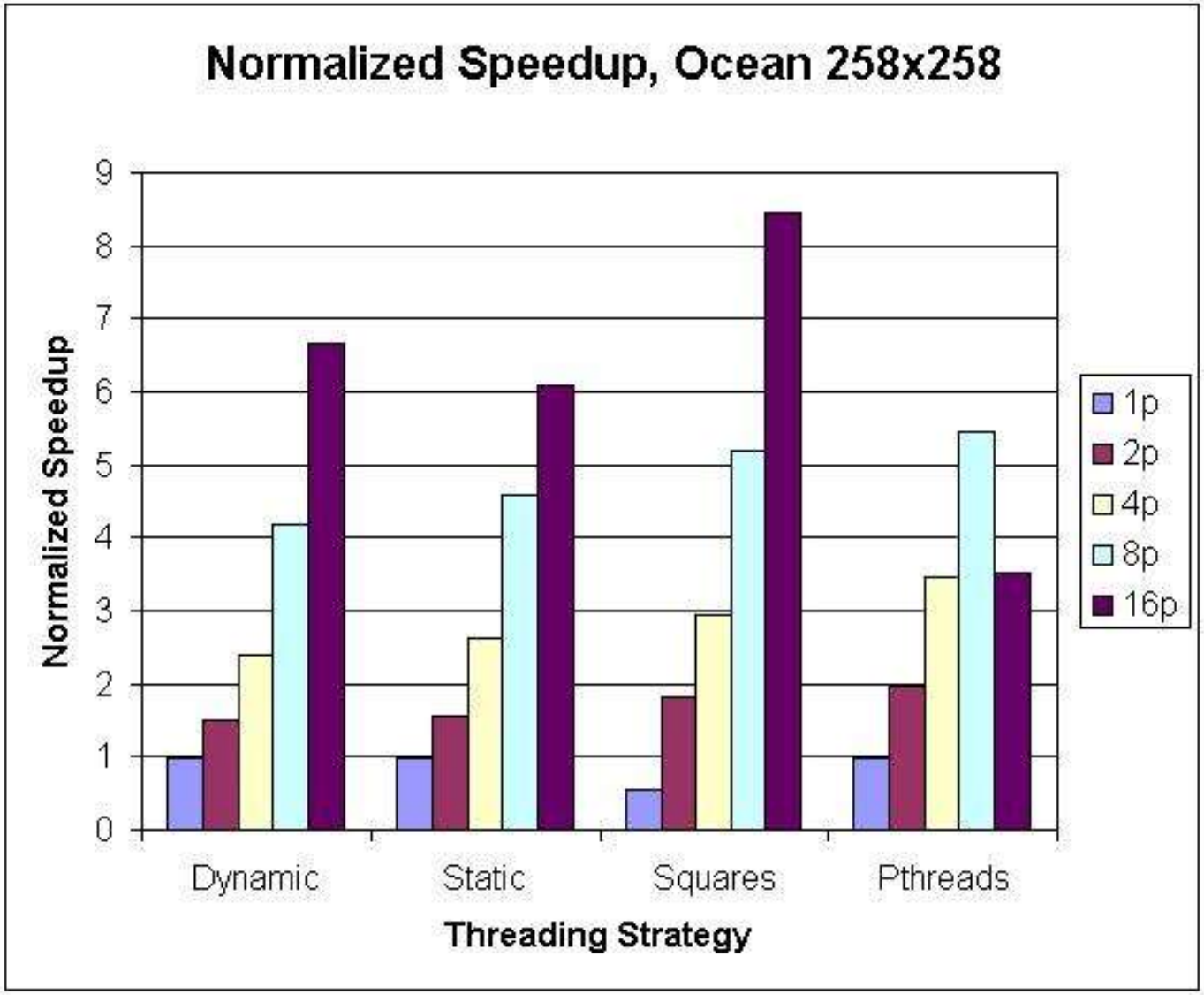
- **`ocean_pthreads` – Each thread traverses a square-shaped section of the ocean. Loop bounds for each thread are determined explicitly.**



Microbenchmark: Ocean



Microbenchmark: Ocean



Evaluation

- OpenMP scales to 16-processor systems
  - Was overhead too high?
    - In some cases, yes
  - Did compiler-generated code compare to hand-written code?
    - Yes!
  - How did the loop scheduling options affect performance?
    - `dynamic` or `guided` scheduling helps loops with variable interation runtimes
    - `static` or predicated scheduling more appropriate for shorter loops
- Is OpenMP the right tool to parallelize scientific application?

Limitations

- OpenMP Requires compiler support
  - Sun Studio compiler
  - Intel VTune
  - Polaris/OpenMP (Purdue)
- OpenMP does not parallelize dependencies
  - Often does not *detect* dependencies
  - Nasty race conditions still exist!
- OpenMP is not guaranteed to divide work optimally among threads
  - Programmer-tweakable with scheduling clauses
  - Still lots of rope available

Limitations

- Doesn't totally hide concept of `volatile` data
  - From a high-level, use of OMP's locks can seem like consistency violations if `flush` directive is forgotten
- Workload applicability
  - Easy to parallelize "scientific" applications
  - How might one create an OpenMP web server? Database?
- Adoption hurdle
  - Search www.sourceforge.net for "OpenMP":
    - 3 results (out of 72,000)

Summary

- OpenMP is a compiler-based technique to create concurrent code from (mostly) serial code
- OpenMP can enable (easy) parallelization of loop-based code
  - Lightweight syntactic language extensions
- OpenMP performs comparably to manually-coded threading
  - Scalable
  - Portable
- Not a silver bullet for all applications

More Information

- www.openmp.org
  - OpenMP official site
- www.llnl.gov/computing/tutorials/openMP/
  - A handy OpenMP tutorial
- www.nersc.gov/nusers/help/tutorials/openmp/
  - Another OpenMP tutorial and reference

Consistency Violation?

```
#pragma omp parallel for \
  shared(x) private(i)
for( i=0; i<100; i++ ) {
  #pragma omp atomic
  x++;
}
printf("%i",x);
```

100

```
#pragma omp parallel for \
  shared(x) private(i)
for( i=0; i<100; i++ ) {
  omp_set_lock(my_lock);
  x++;
  #pragma omp flush
  omp_unset_lock(my_lock);
}
printf("%i",x);
```

1900

OpenMP Syntax

- General syntax for OpenMP directives

```
#pragma omp directive [clause…] CR
```

- *Directive* specifies type of OpenMP operation
  - Parallelization
  - Synchronization
  - Etc.
- *Clauses* (optional) modify semantics of *Directive*

OpenMP Syntax

- PARALLEL syntax

```
#pragma omp parallel [clause…] CR
      structured_block
```

Ex:

```
#pragma omp parallel
{
  printf("Hello!\n");
} // implicit barrier
```

Output: (T=4)

```
Hello!
Hello!
Hello!
Hello!
```

OpenMP Syntax

- DO/for Syntax (DO-Fortran, for-C)

```
#pragma omp for [clause…] CR
      for_loop
```

Ex:

```
#pragma omp parallel
{
   #pragma omp for private(i) shared(x) \
                   schedule(static,x/N)
   for(i=0;i<x;i++) printf("Hello!\n");
} // implicit barrier
```

Note: Must reside inside a `parallel` section

OpenMP Syntax

More on Clauses

- `private()` – A variable in private list is private to each thread
- `shared()` – Variables in shared list are visible to all threads
  - Implies no synchronization, or even consistency!
- `schedule()` – Determines how iterations will be divided among threads
  - `schedule(static, C)` – Each thread will be given C iterations
    - Usually T*C = Number of total iterations
  - `schedule(dynamic)` – Each thread will be given additional iterations as-needed
    - Often less efficient than considered static allocation
- `nowait` – Removes implicit barrier from end of block

OpenMP Syntax

- PARALLEL FOR (combines `parallel` and `for`)

```
#pragma omp parallel for [clause…] CR
      for_loop
```

Ex:

```
#pragma omp parallel for shared(x)\
                         private(i) \

  schedule(dynamic)
for(i=0;i<x;i++) {
  printf("Hello!\n");
}
```

OpenMP Syntax

- ATOMIC syntax

```
#pragma omp atomic CR
      simple_statement
```

Ex:

```
#pragma omp parallel shared(x)
{
   #pragma omp atomic
   x++;
} // implicit barrier
```

OpenMP Syntax

- **CRITICAL syntax**

```
#pragma omp critical CR
      structured_block
```

Ex:

```
#pragma omp parallel shared(x)
{
   #pragma omp critical
   {
     // only one thread in here
   }
} // implicit barrier
```

OpenMP Syntax

ATOMIC vs. CRITICAL

- Use ATOMIC for “simple statements”
  - Can have lower overhead than CRITICAL if HW atomics are leveraged (implementation dep.)
- Use CRITICAL for larger expressions
  - May involve an unseen implicit lock

OpenMP Syntax

- MASTER – only Thread 0 executes a block

```
#pragma omp master CR
      structured_block
```

- SINGLE – only one thread executes a block

```
#pragma omp single CR
      structured_block
```

- No implied synchronization

OpenMP Syntax

- BARRIER

```
#pragma omp barrier CR
```

- Locks
  - Locks are provided through `omp.h` library calls
  - `omp_init_lock()`
  - `omp_destroy_lock()`
  - `omp_test_lock()`
  - `omp_set_lock()`
  - `omp_unset_lock()`

OpenMP Syntax

- FLUSH

```
#pragma omp flush CR
```

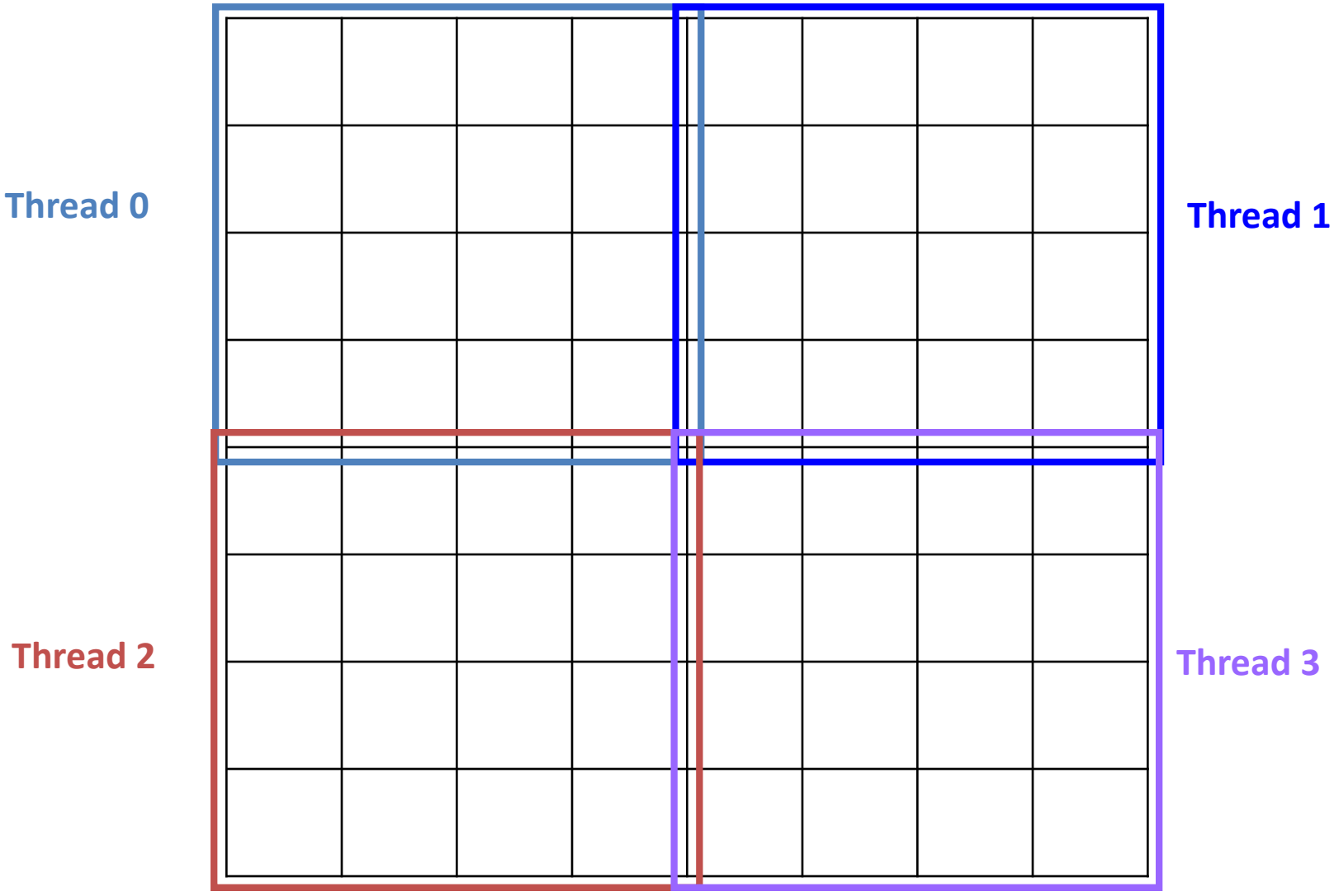
- Guarantees that threads' views of memory is consistent
- Why? Recall OpenMP directives…
  - Code is generated by directives at compile-time
    - Variables are not always declared as `volatile`
    - Using variables from registers instead of memory can seem like a consistency violation
  - Synch. Often has an implicit flush
    - ATOMIC, CRITICAL

OpenMP Syntax

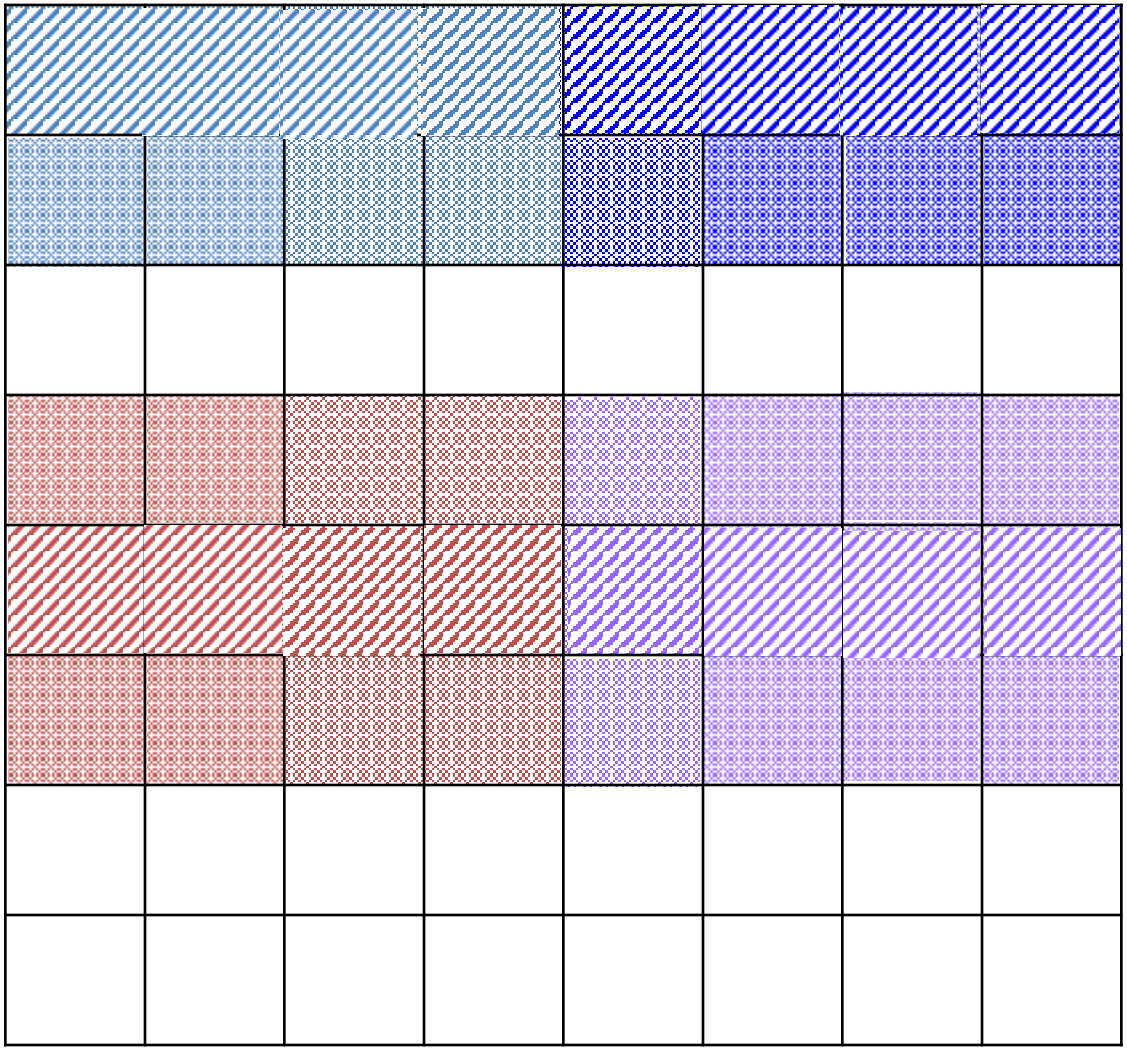
- Functions

```
omp_set_num_threads()
omp_get_num_threads()
omp_get_max_threads()
omp_get_num_procs()
omp_get_thread_num()
omp_set_dynamic()
omp_[init destroy test set
unset]_lock()
```

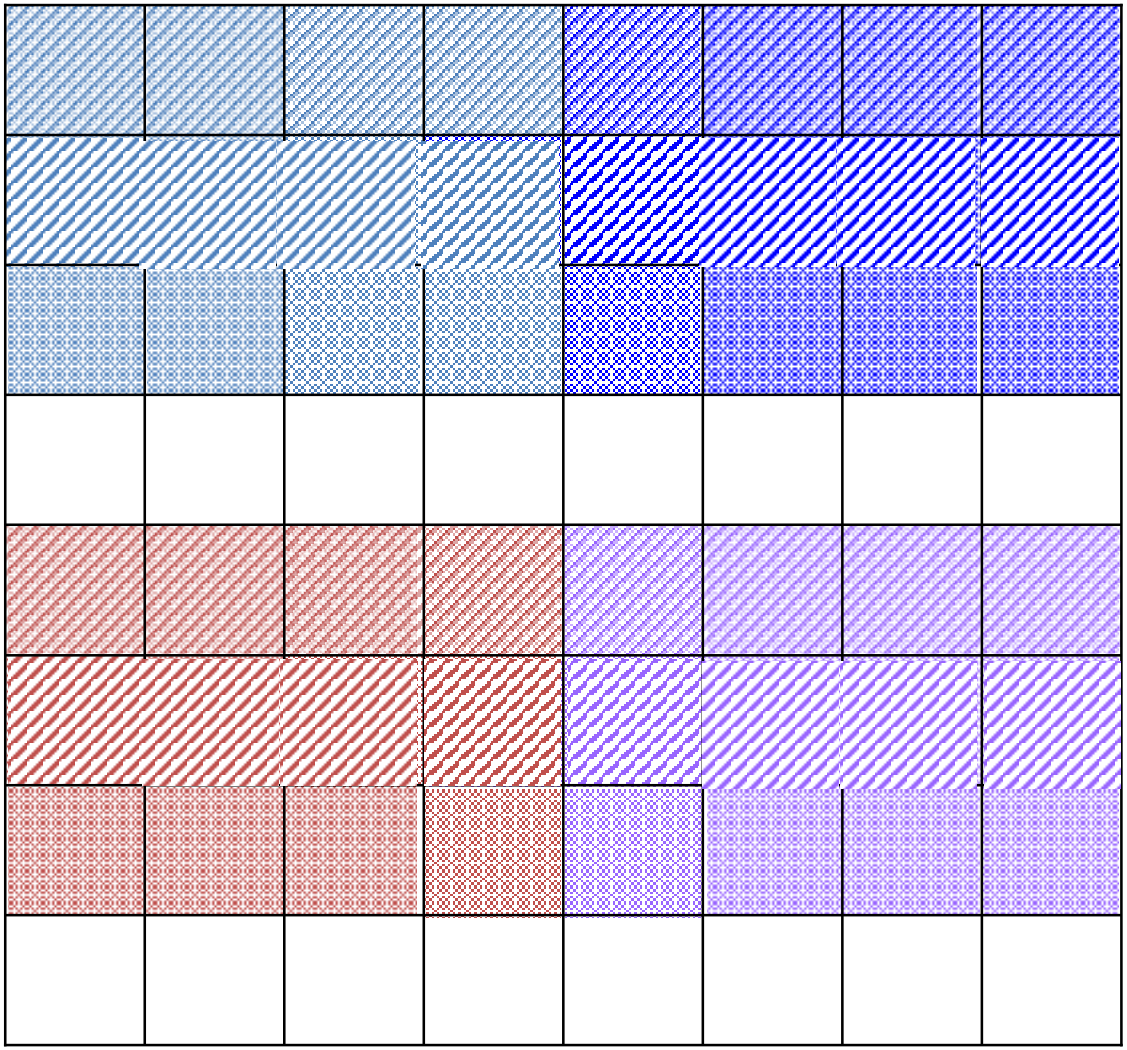
Microbenchmark: Ocean



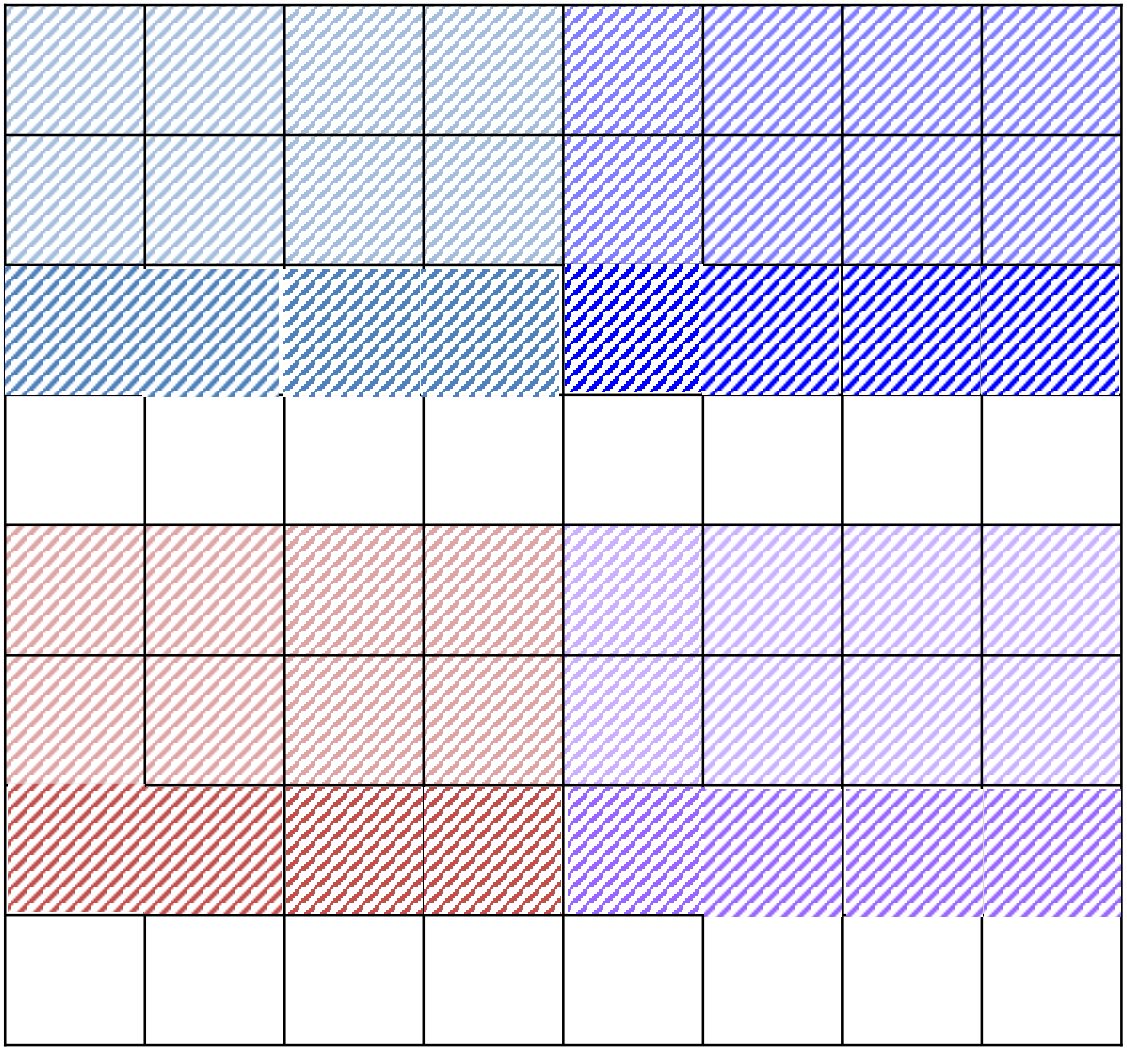
Microbenchmark: Ocean



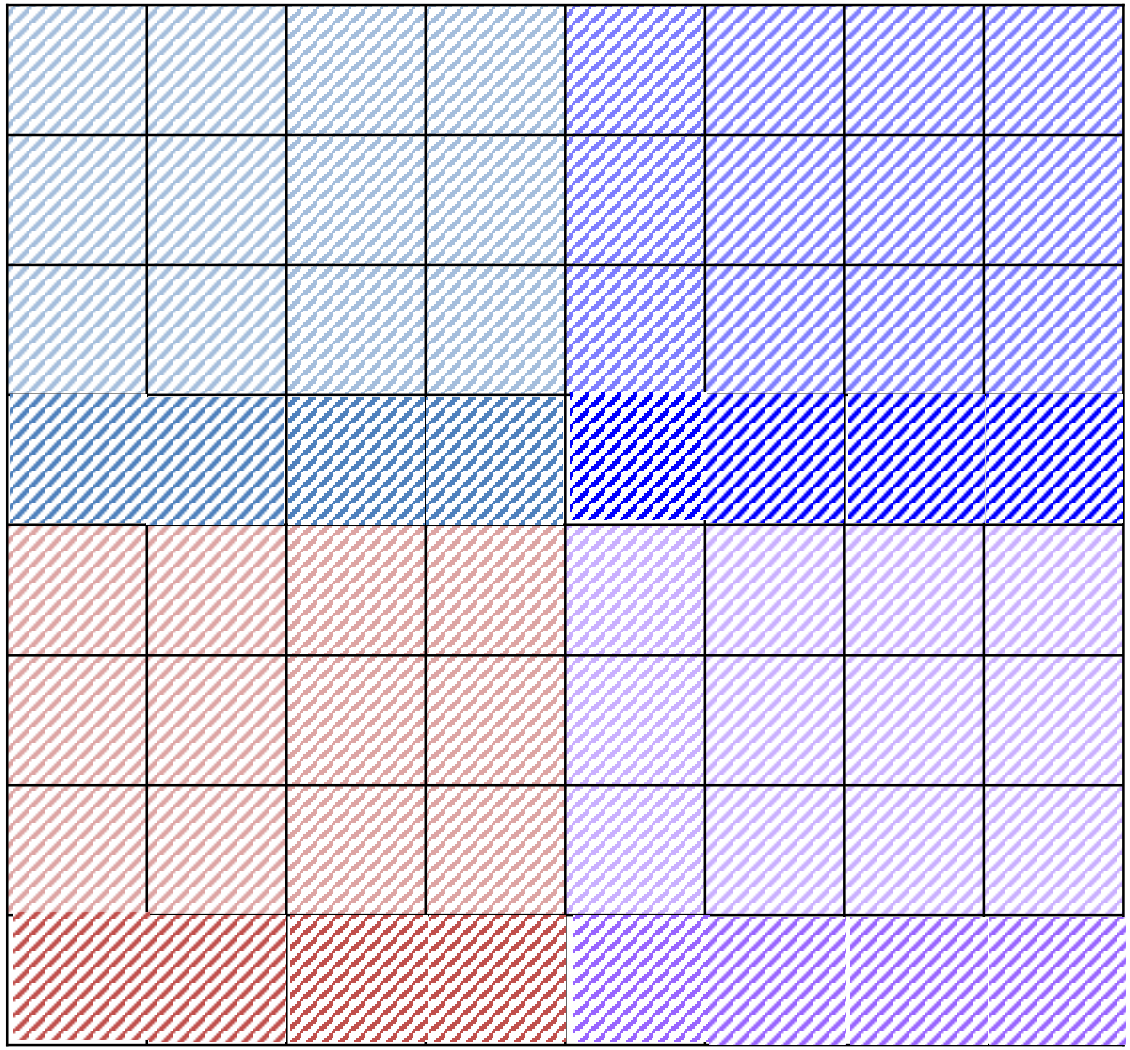
Microbenchmark: Ocean



Microbenchmark: Ocean



Microbenchmark: Ocean



Microbenchmark: Ocean