Bucknell UNIVERSITY

Threads

CSCI 315 Operating Systems Design
Department of Computer Science

Notice: The slides for this lecture have been largely based on those accompanying the textbook *Operating Systems Concepts*, 9th ed., by Silberschatz, Galvin, and Gagne, Prof. Xiannong Meng's slides, and Blaise Barney (LLNL) "POSIX Threads Programming" online tutorial.

Interlude

Pointer Recap

```
NAME
       wait, waitpid, waitid - wait for process to change state

SYNOPSIS
       #include <sys/types.h>
       #include <sys/wait.h>

       pid_t wait(int *status);

       pid_t waitpid(pid_t pid, int *status, int options);
```

Pointer Recap

```
int ret_val;
.
.
.
ret_val = wait( –???– );
.
.
.
```

Pointer Recap

```
int ret_val;
int *status;
.
.
.
ret_val = wait(status);
.
.
.
```

```
int ret_val;
int status;
.
.
.
ret_val = wait(&status);
.
.
.
```

- Do both options compile correctly?
- Do both options run correctly?
- Can you explain what each one does?

Pointer Recap 2

```
char *c;
.
.
.
*c = 'a';
.
.
.
```

- Do this compile correctly?
- Do this run correctly?

Pointer Recap 2

```
char *c;
.
.
.
*c = 'a';
.
.
.
```

```
char *c = malloc(10);
.
.
.
*c = 'a';
.
.
.
```

What is the difference between the two?

Pointer Recap 2

```
char *c = malloc(10);
.
.
.
*c = 'a';
.
.
.
```

```
char *c = malloc(10);
.
.
.
*c = 'a';
c[1] = c[0];
*(c+2) = c[1];
.
.
```

- What is the value of c[1] after the assignment?
- What is the value of c[2] after the assignment?

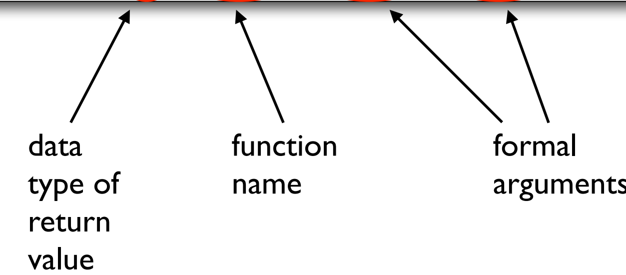
Function Recap

```
int summation(int start, int end);
```

Function Recap

Function prototype

```
int summation(int start, int end);
```



Function Recap

Function prototype

```
int summation(int start, int end);
```

What is this???

```
int *f(int, int);
```

Function Pointer Recap

Function prototype

```
int summation(int start, int end);
```

Function pointer declaration

```
int *f(int, int);
```

Function pointer assignment

```
f = summation;
```

Function Pointer Parameter

Function prototype

```
int compute(int, int, int *g(int, int);
```

Function body

```
int compute(int a, int b, int *g(int, int) {
  return g(a, b);
}
```

Function Recap

Function prototype

```
int summation(int start, int end);
```

data type of return value — function name — formal arguments

And now, our main attraction…

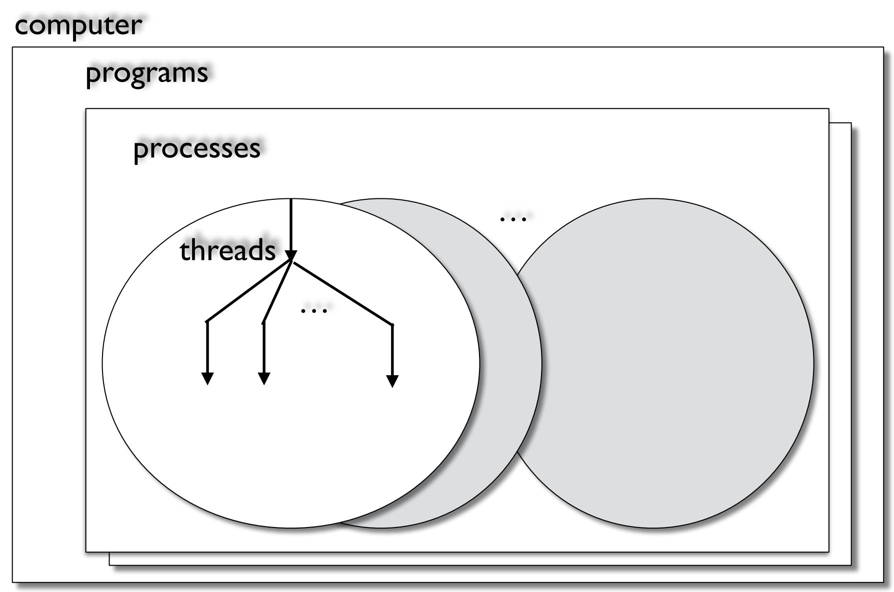
Motivation

- Process level concurrency is often not enough.
- One **process** may contain multiple **threads.**
- Many modern applications are multithreaded.
- **Different tasks** within the application can be implemented by **different threads**: update display, fetch data, check spelling, service a network request.
- Process creation is time consuming, thread creation is not.
- Threads can simplify coding and increase efficiency.
- OS Kernels are generally multithreaded. OS and/or libraries have support for user-level threads.

More Motivation?

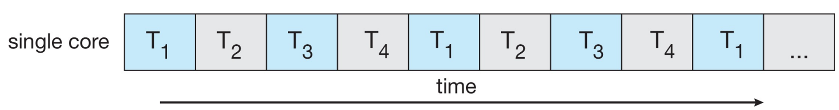
- **Responsiveness:** multiple threads can be executed in parallel (in multi-core machines)
- **Resource sharing:** multiple threads have access to the same data, sharing made easier
- **Economy:** the overhead in creating and managing threads is smaller
- **Scalability:** more processors (or cores), more threads running in parallel

Applications: A Hierarchical View

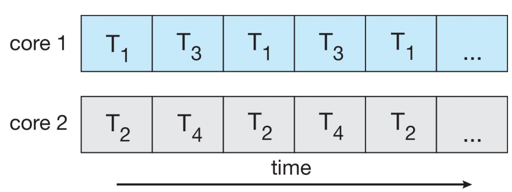


Concurrency and Parallelism

Concurrent execution in single-core system



Parallelism on multi-core system



Look at `pthread_create(3)`

```
NAME
       pthread_create - create a new thread

SYNOPSIS
       #include <pthread.h>

       int pthread_create(pthread_t *thread,
                          const pthread_attr_t *attr,
                          void *(*start_routine) (void *),
                          void *arg);

       Compile and link with -pthread.
```

Explain:
(a) what `void *p;` means
(b) what this means: `void *(*start_routine) (void *)`

Here's the code for my thread:

```
void *sleeping(void *arg) {
  int sleep_time = (int)arg;
  printf("thread %ld sleeping %d seconds ...\n",
  pthread_self(), sleep_time);
  sleep(sleep_time);
  printf("\nthread %ld awakening\n", pthread_self());
  return (NULL);
}
```

OK, how do I understand this?

```
void *sleeping(void *arg) {
  int sleep_time = (int)arg;
  printf("thread %ld sleeping %d seconds ...\n",
  pthread_self(), sleep_time);
  sleep(sleep_time);
  printf("\nthread %ld awakening\n", pthread_self());
  return (NULL);
}
```

Creating five identical threads

```
/* COMPILE WITH: gcc thread-ex.c -lpthread -o thread-ex */
#include <stdio.h>
#include <pthread.h>
#define NUM_THREADS 5
#define SLEEP_TIME 3

void *sleeping(void *); /* forward declaration to thread routine */

int main(int argc, char *argv[]) {
int i;
pthread_t tid[NUM_THREADS]; /* array of thread IDs */
for ( i = 0; i < NUM_THREADS; i++)
  pthread_create(&tid[i], NULL, sleeping,(void *)SLEEP_TIME);

for ( i = 0; i < NUM_THREADS; i++)
  pthread_join(tid[i], NULL);

printf("main() reporting that all %d threads have terminated\n", i);
return (0);
} /* main */
```

So, threads can't take parameters and can't return anything?

```
void * sleeping(void *arg) {
  int sleep_time = (int)arg;
  printf("thread %ld sleeping %d seconds ...\n",
  pthread_self(), sleep_time);
  sleep(sleep_time);
  printf("\nthread %ld awakening\n", pthread_self());
  return (NULL);
}
```

A thread can take parameter(s) pointed by its **`arg`** and can return a pointer to some memory location that stores its results. Gotta be careful with these pointers!!!

Passing arguments into thread

```
pthread_t tid[NUM_THREADS]; /* array of thre    IDs */
for ( i = 0; i < NUM_THREADS; i++)
  pthread_create(&tid[i], NULL, sleeping,(void *)SLEEP_TIME);

...
```

- Casting is powerful, so it deserves to be used carefully
- This is disguising an integer as a `void *` **(a hack?)**
- Have to remove the disguise inside the thread routine

Passing arguments into thread

```
struct args_t {
  int id;
  char *str;
} myargs[NUM_THREADS];

void * thingie(void *arg) {
  struct args_t *p = (struct args_t*) arg;
  printf("thread id= %d, message= %s\n", p->id, p->msg);
}
```

```
for ( i = 0; i < NUM_THREADS; i++)
  pthread_create(&tid[i], NULL, thingie,(void *)&myargs[i]);

...
```

Passing results out of thread

```
struct args_t {
  int id;
  char *str;
  double result;
} myargs[NUM_THREADS];

void * thingie(void *arg) {
  struct args_t *p = (struct args_t*) arg;
  printf("thread id= %d, message= %s\n", p->id, p->msg);
  p->result = 3.1415926 * p->id;
  return(NULL); // or return(arg)
}
```

Option 1

Passing results out of thread

```
struct args_t {
  int id;
  char *str;
} myargs[NUM_THREADS];

struct results_t {
  double result;
};

void * thingie(void *arg) {
  struct args_t *p = (struct args_t*) arg;
  struct results_t *r = malloc(sizeof(struct results_t));

  printf("thread id= %d, message= %s\n", p->id, p->msg);
  r->result = 3.1415926 * arg->id;
  return((void*) r);
}
```

Watch out for **memory leaks!**

Option 2

Your thread returns a `void *`

What is the point of returning this value?

Look at `pthread_join(3)`

```
NAME
       pthread_join - join with a terminated thread

SYNOPSIS
       #include <pthread.h>

       int pthread_join(pthread_t thread, void **retval);
```

Analogous to **`wait(2)`** and **`waitpid(2)`**

```
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

Look at `pthread_join(3)`

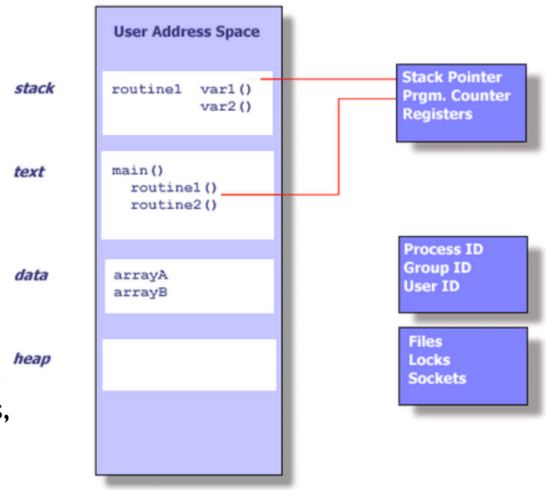
```
NAME
       pthread_join - join with a terminated thread

SYNOPSIS
       #include <pthread.h>

       int pthread_join(pthread_t thread, void **retval);
```

A pointer to a pointer to something

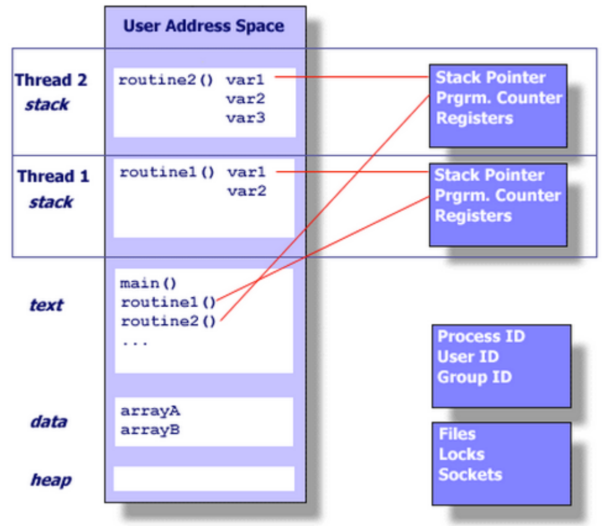
Threads and Processes

Process

Process ID,
process group ID,
user ID, group ID,
Environment,
Program instructions,
Registers,
Stack,
Heap,
File descriptors,
Signal actions,
Shared libraries,
IPC message queues, pipes, semaphores, or shared memory).

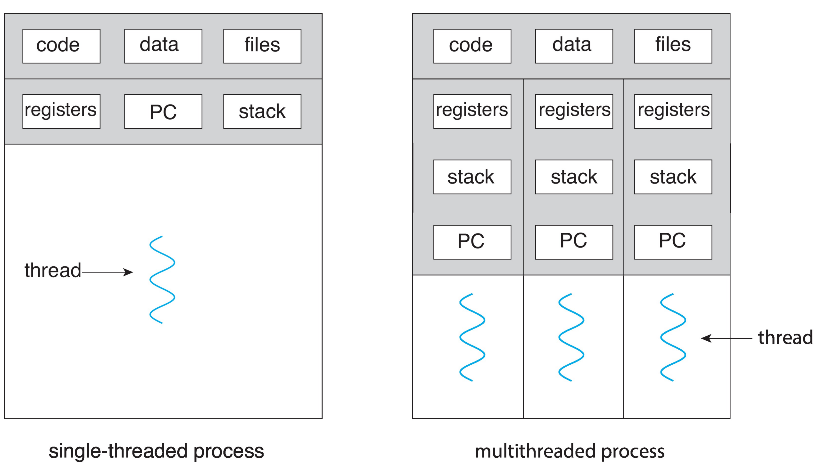


Thread

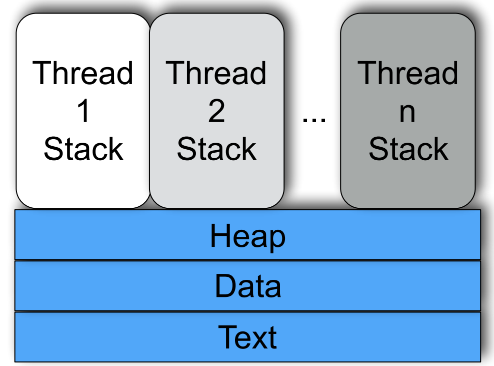


Stack pointer
Registers
Scheduling properties (such as policy or priority)
Set of pending and blocked signals
Thread specific data

Thread

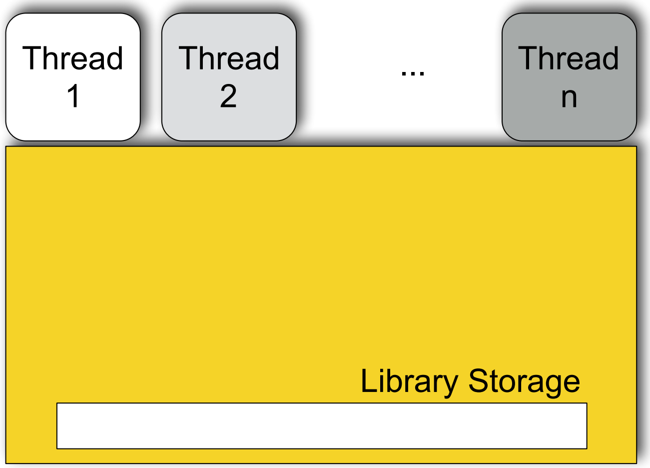


Shared Memory Model

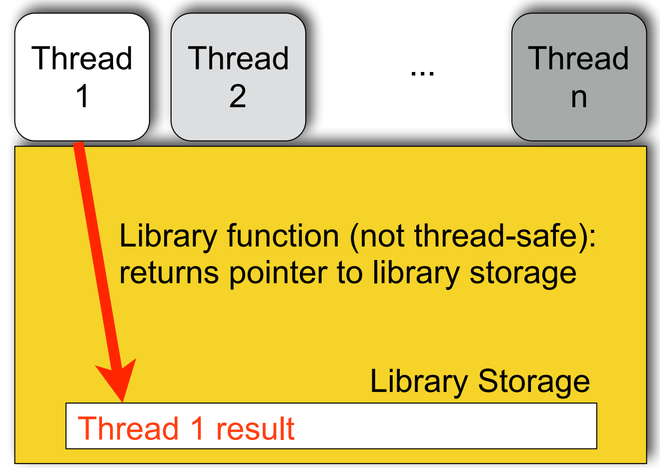


- All threads have access to the same global, shared memory
- Threads also have their own private data (how?)
- Programmers are responsible for protecting globally shared data

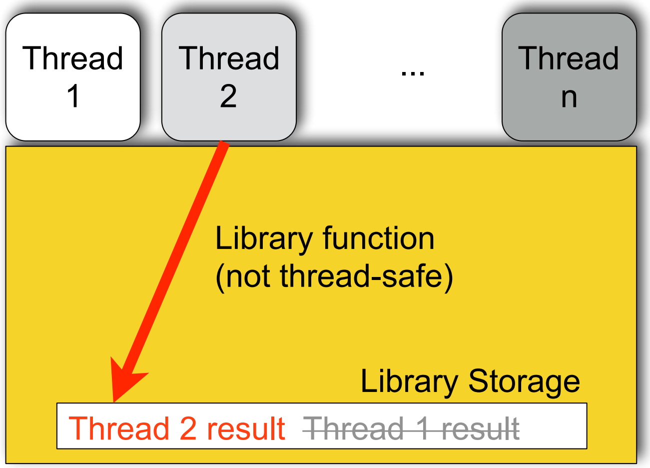
Thread Safety



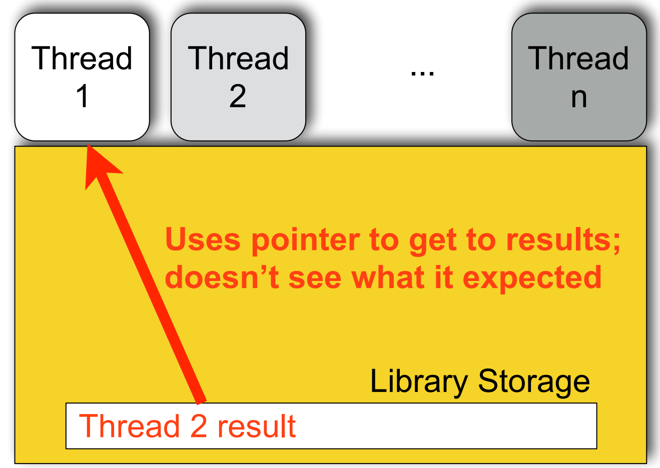
Thread Safety



Thread Safety



Thread Safety



Thinking about Performance

Speedup

If you care about performance, your speed up needs to be bigger than 1. (If it's not, you have a problem.) But you need to be honest!

$$\text{speedup} = \frac{\text{time of the best sequential solution}}{\text{time of the parallel solution}}$$

Amdhal's Law

$$\text{speedup} \leq \frac{1}{S + \frac{(1-S)}{N}}$$

S = portion that must execute serially

(1-S) = portion that can be parallelized

N = number of cores

AMDAHL'S LAW

SPEED UP BOUND

SERIAL (SEC)	PARALLEL (SEC)
0.001	0.999
0.005	0.995
0.01	0.99
0.05	0.95
0.1	0.9
0.5	0.5

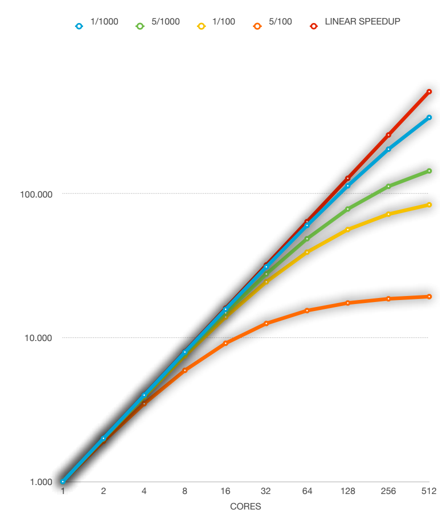
CORES	SPEEDUP BOUND
1	1.000
2	1.998
4	3.988
8	7.944
16	15.764
32	31.038
64	60.207
128	113.576
256	203.984
512	338.848

CORES	SPEEDUP BOUND
1	1.000
2	1.990
4	3.941
8	7.729
16	14.884
32	27.706
64	48.669
128	78.287
256	112.527
512	144.023

CORES	SPEEDUP BOUND
1	1.000
2	1.980
4	3.883
8	7.477
16	13.913
32	24.427
64	39.264
128	56.388
256	72.113
512	83.797

CORES	SPEEDUP BOUND
1	1.000
2	1.905
4	3.478
8	5.926
16	9.143
32	12.549
64	15.422
128	17.415
256	18.618
512	19.284

CORES	LINEAR SPEEDUP
1	1.000
2	2.000
4	4.000
8	8.000
16	16.000
32	32.000
64	64.000
128	128.000
256	256.000
512	512.000



Challenges in Parallel Programming

- Identifying "parallelizable" tasks
- Load balance
- Data decomposition
- Data dependency
- Testing and debugging

Multithreading Models

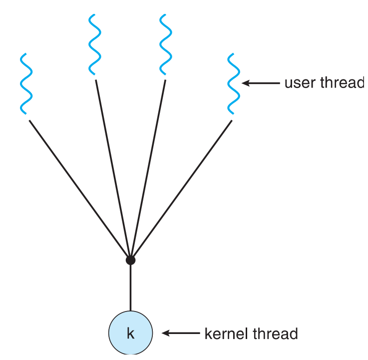
User threads

Managed by a library without kernel support; runs at user level

Kernel threads

Managed directly by the operating system

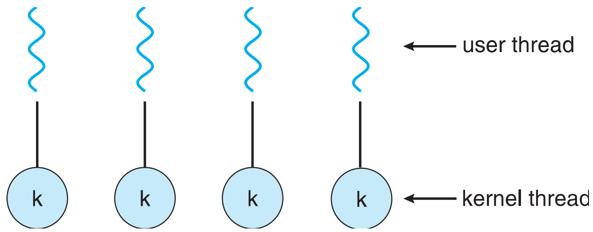
Many-To-One Model



Disadvantages

Advantages

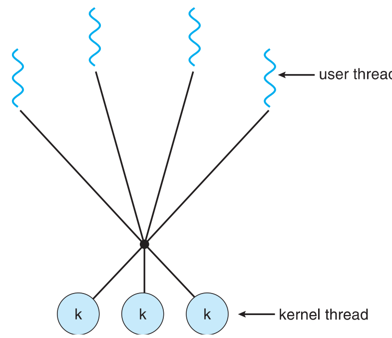
One-To-One Model



Disadvantages

Advantages

Many-To-Many Model



Disadvantages

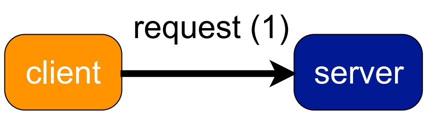
Advantages

What are thread pools?

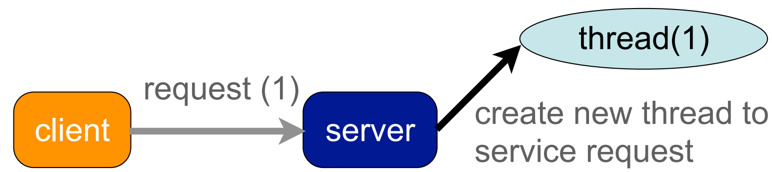
Anything good or bad?

A Typical Application

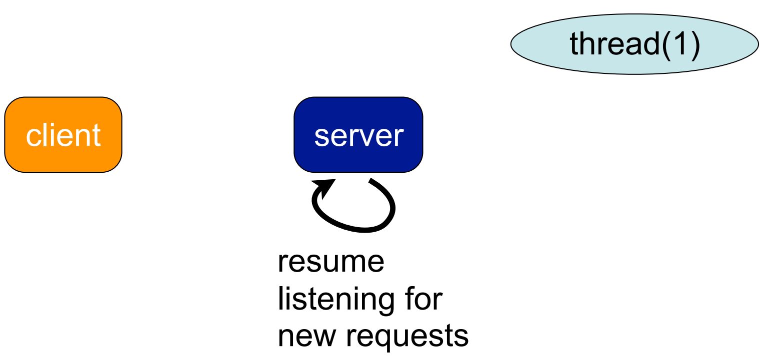
Multithreaded Server Architecture



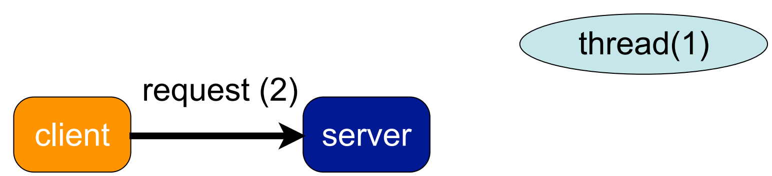
Multithreaded Server Architecture



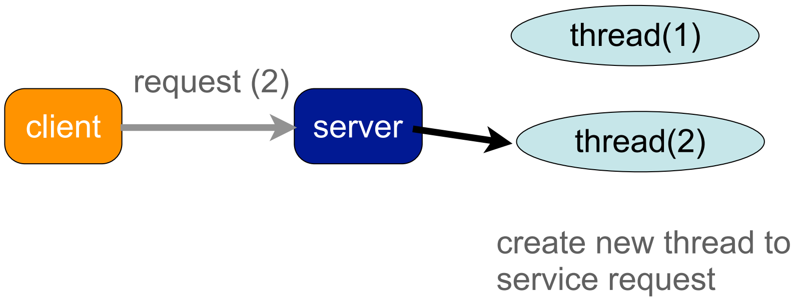
Multithreaded Server Architecture



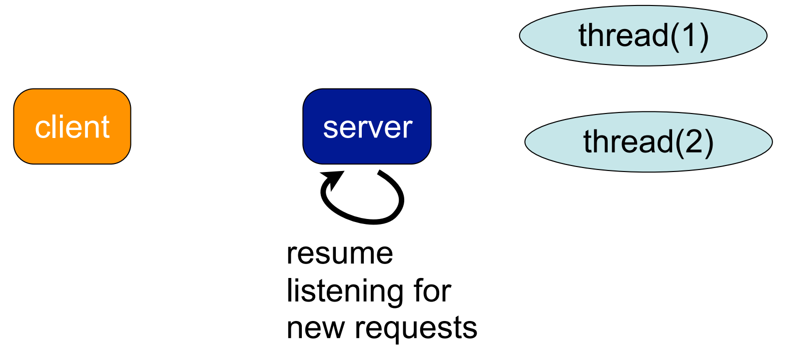
Multithreaded Server Architecture



Multithreaded Server Architecture



Multithreaded Server Architecture



BACK TO

BACK TO

Inter process communication

Inter process communication

- file
- pipe
- shared memory
- message passing
- …

Processes on the same machine

- remote procedure call
- message passing
- sockets
- …

Processes on different machines

Networking

Connectivity

Wish List:

– Interconnect machines.

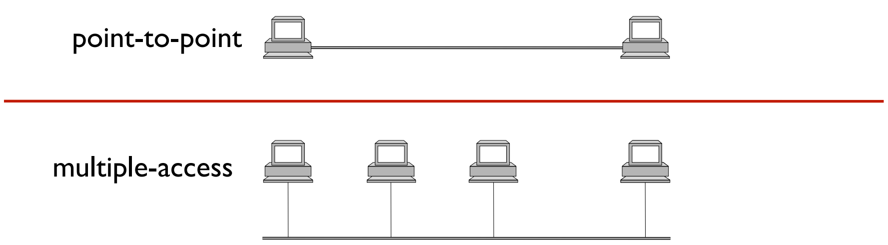
– Maintain data *confidentiality*, data *integrity*, and system *accessibility*.

– Support growth by allowing more and more computers, or nodes, to join in (*scalability*).

– Support increases in geographical coverage.

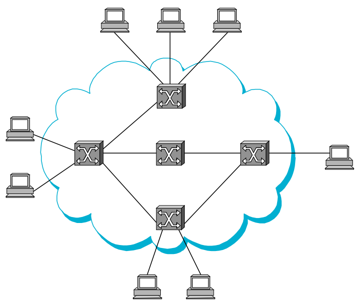
Links

Each node needs one interface (NIC) for each link.

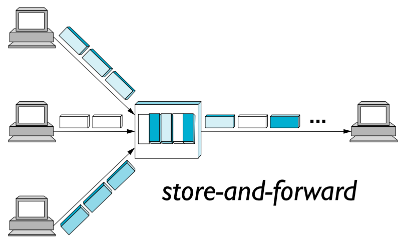


Geographical coverage and scalability are limited.

Switched Networks



Circuit Switched



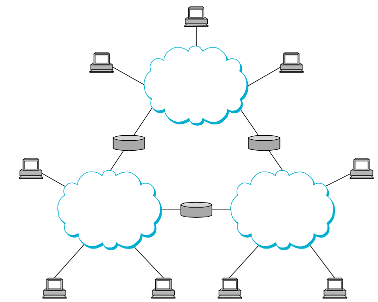
Packet Switched

Internetworking

To interconnect two or more networks, one needs a ***gateway*** or ***router***.

Host-to-host connectivity is only possible if there's a uniform ***addressing*** scheme and a ***routing*** mechanism.

Messages can be sent to a single destination (***unicast***), to multiple destinations (***multicast***), or to all possible destinations (***broadcast***).



The ISO/OSI Reference Model

Source: Computer Networks, Andrew Tanenbaum

ISO: International Standards Organization
OSI: Open Systems Interconnection

The protocol *stack*:

- Application
- Presentation
- Session
- Transport
- Network
- Data link
- Physical

The idea behind the model: Break up the design to make implementation simpler. Each layer has a well-defined function. Layers pass to one another only the information that is relevant at each level. Communication happens only between adjacent layers.

The Layers in the ISO/OSI RF Model

Physical: Transmit raw bits over the medium.

Data Link: Implements the abstraction of an error free medium (handle losses, duplication, errors, flow control).

Network: Routing.

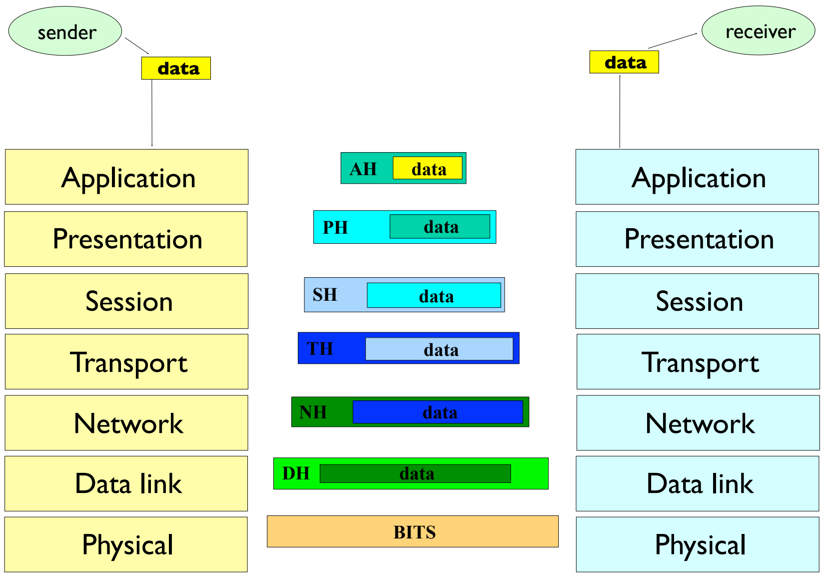
Transport: Break up data into chunks, send them down the protocol stack, receive chunks, put them in the right order, pass them up.

Session: Establish connections between different users and different hosts.

Presentation: Handle syntax and semantics of the info, such as encoding, encrypting.

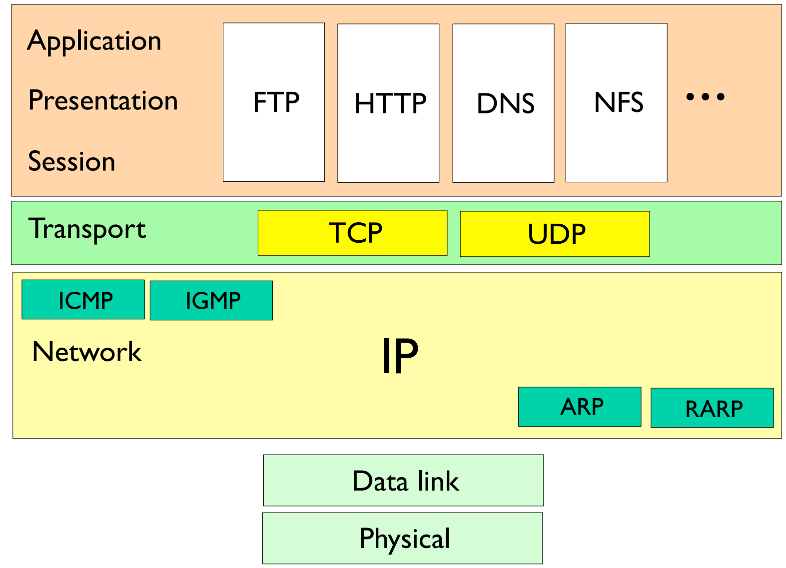
Application: Protocols commonly needed by applications (cddb, http, ftp, telnet, etc).

Communication Between Layers in Different Hosts



The Layers in the TCP/IP Protocol Suite

Source: The TCP/IP Protocol Suite, Behrouz A. Forouzan



Socket Functions