Lecture 8

Communication between Programs
&
Concurrent Programming

see: 第三部分：程序间的交互与通信

Shutdown Mobile...

Key Point

- Threads are a popular and useful tool for introducing concurrency in programs.
- Threads are typically more efficient than processes, and it is much easier to share data between threads than between processes.
- However, the ease of sharing introduces the possibility of synchronization errors that are difficult to diagnose.
- Programmers writing threaded programs must be careful to protect shared data with the appropriate synchronization mechanisms.
- Functions called by threads must be thread-safe. Races and deadlocks must be avoided.
- Wise programmer approaches the design of threaded programs with great care and not a little trepidation

Concurrency

Why concurrent programming

* concurrency on apps
  * parallel computing on multi-processors
  * visit slow I/O devices
  * interact with users
  * delay working for reduction of processing (coalescing)
  * serve multiple network clients

Profile

Process

	Kernel I/O	Computing
Sync (Single Process)	—	Traditional Programming Php / Nginx
Async (Multiple Process)	Multiplexing	Async Programming Node.js / V8
Multiple Thread (Multiple Process) (Multiple Threads)	Node Nginx	Apache

Concurrent Programming

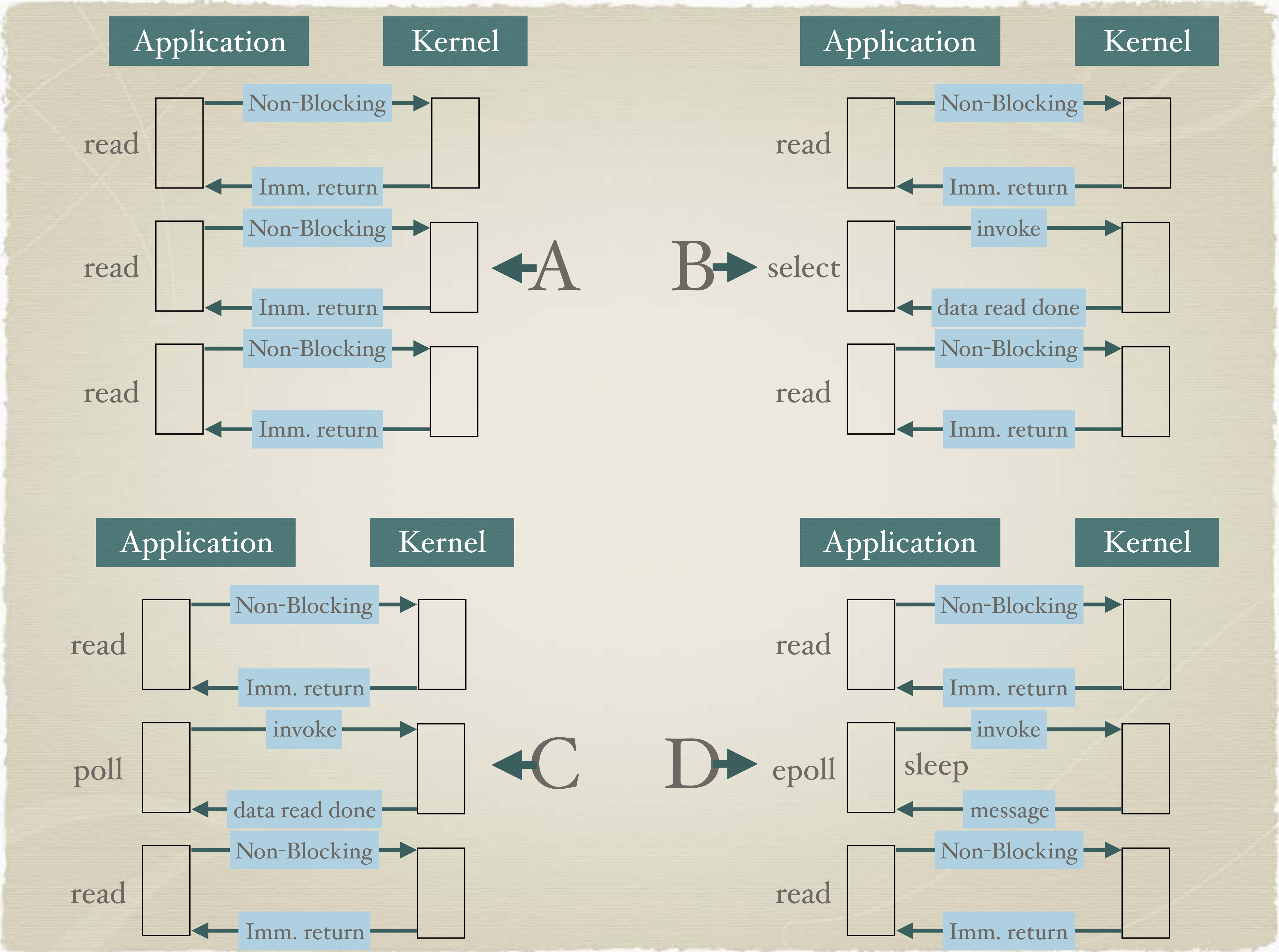
Some Terms

* Sync/Async
* Multiple process/ Multiple thread
* Concurrent programming
* Async I/O / Blocking I/O / Non-Blocking I/O
* Callback / Event

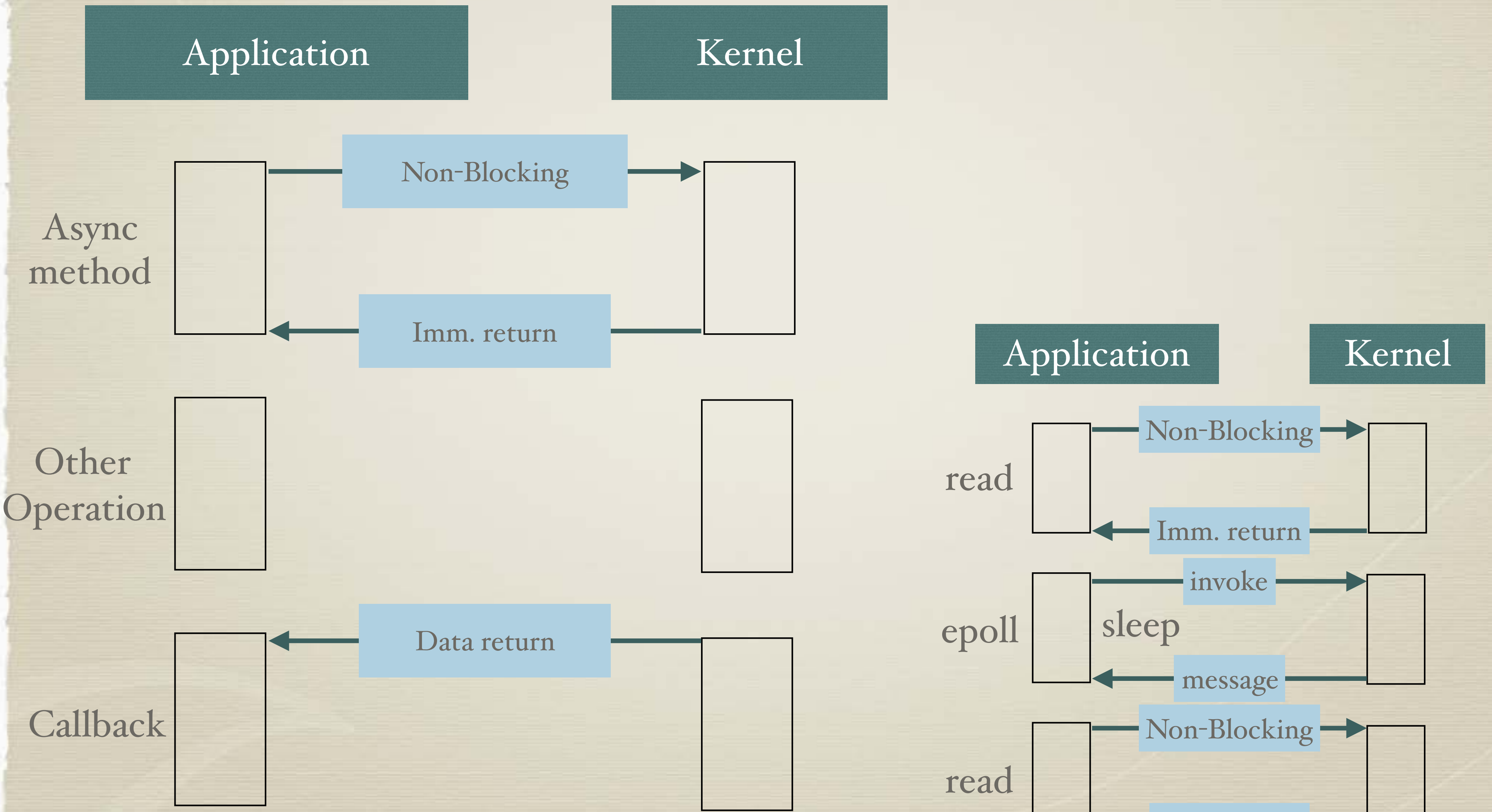
I/O type	CPU circle cost
CPU L1 Cache	3
CPU L2 Cache	14
Main memory	250
Hard disk	41,000,000
Network	240,000,000

Manners of concurrent programming

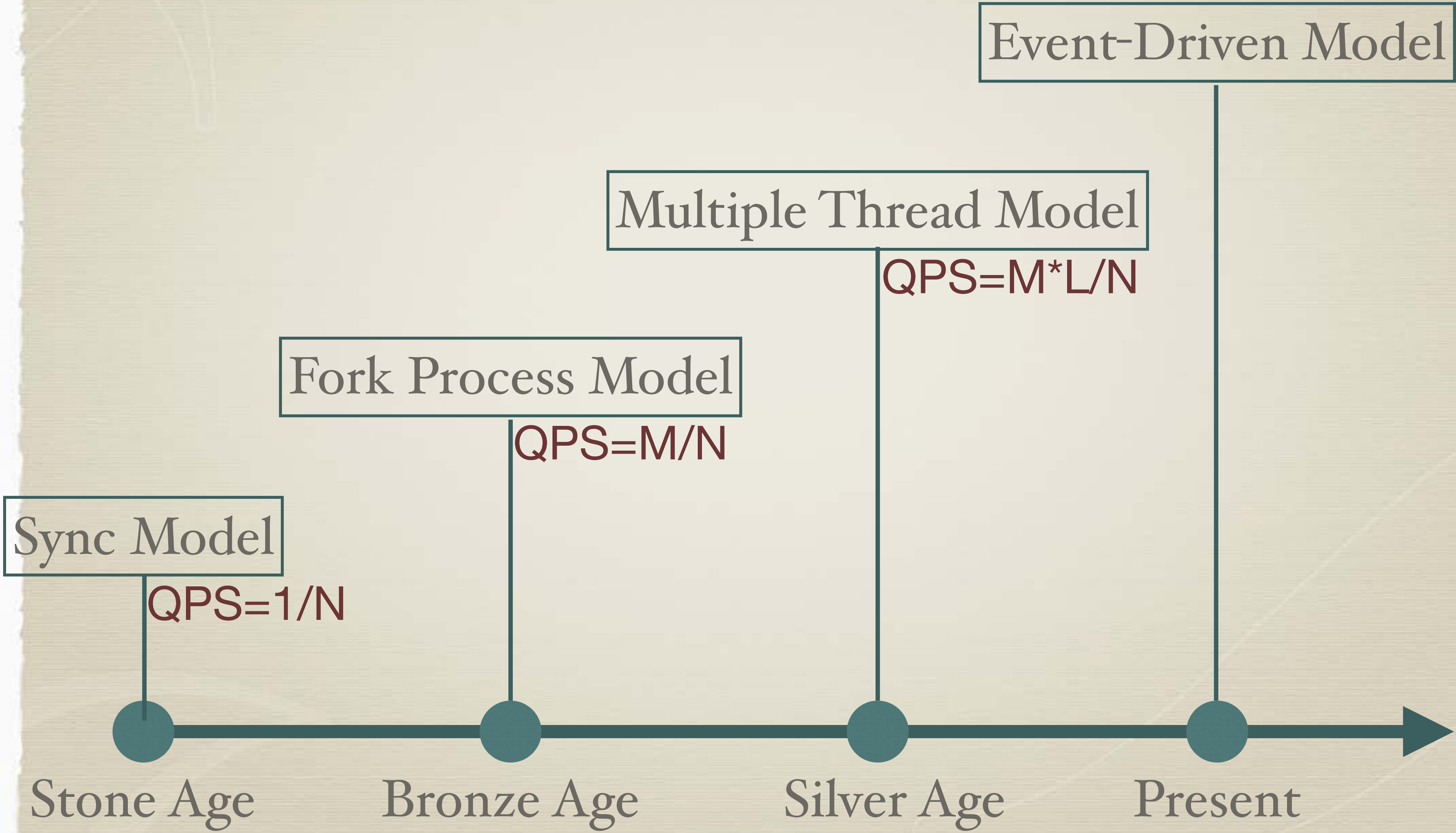
* Process
* I/O multiplexing
* Threads


Application
Kernel
read
Non-Blocking
Imm. return
read
Non-Blocking
Imm. return
read
Non-Blocking
Imm. return
A
Application
Kernel
read
Non-Blocking
Imm. return
B
select
invoke
data read done
read
Non-Blocking
Imm. return
Application
Kernel
read
Non-Blocking
Imm. return
poll
invoke
data read done
read
Non-Blocking
Imm. return
C
Application
Kernel
read
Non-Blocking
Imm. return
D
epoll
invoke
sleep
message
read
Non-Blocking
Imm. return

Ideal Non-Blocking I/O



Client-Server Models

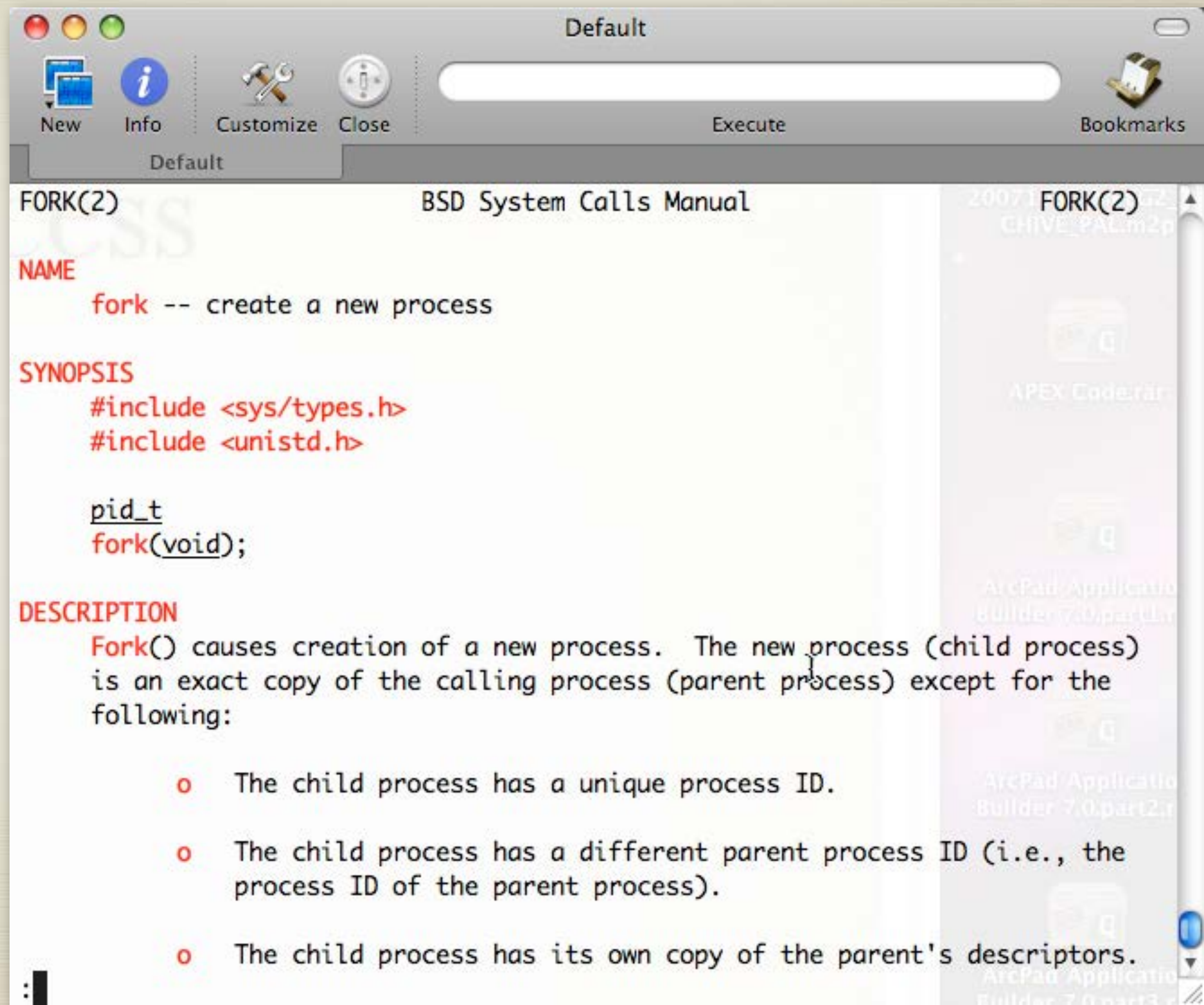


Process

* Process is the simplest way of concurrent programming
    * fork()
    * exec()
    * waitpid()

Process

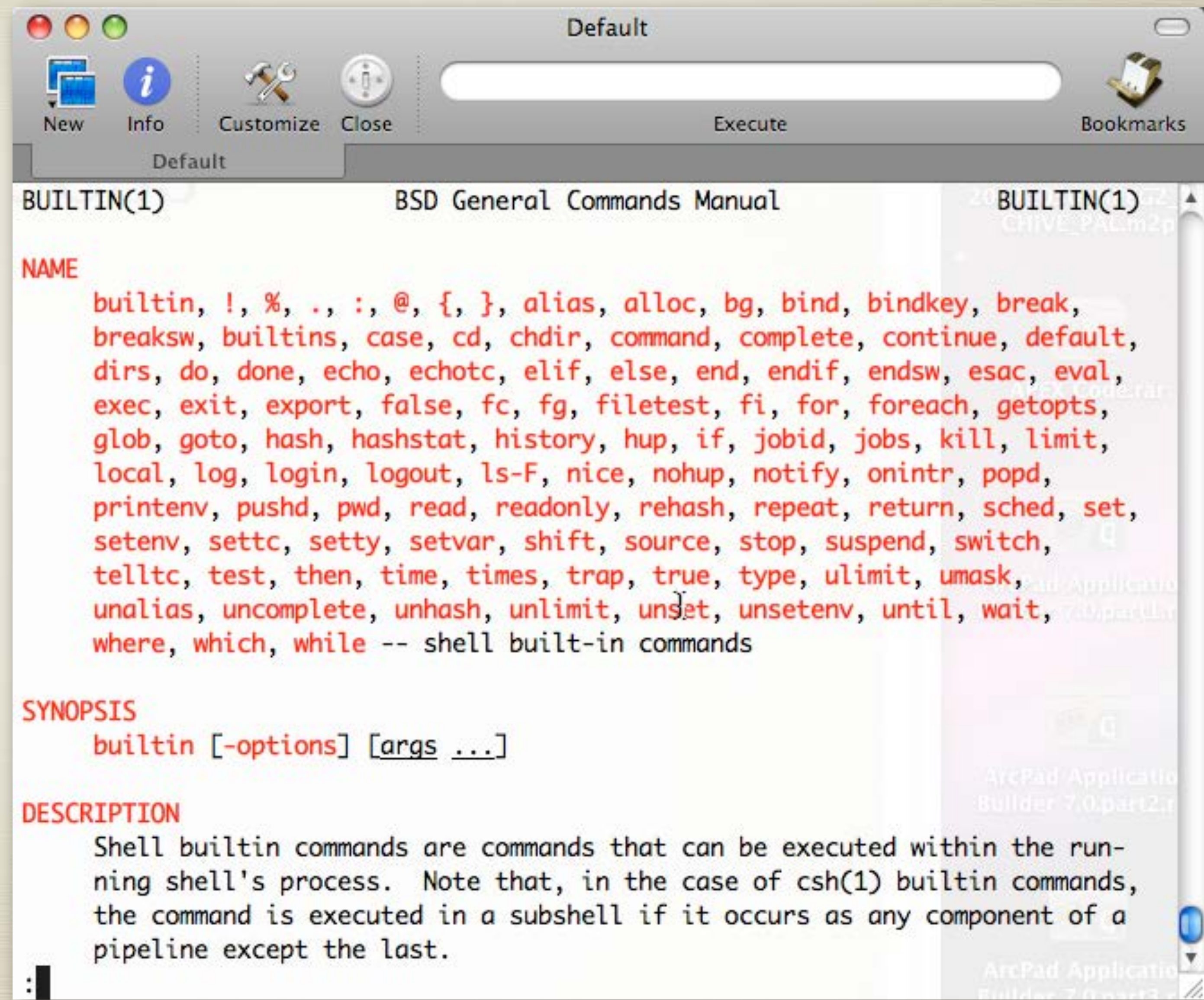
* fork()




```
FORK(2)                 BSD System Calls Manual                 FORK(2)

NAME
     fork -- create a new process

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

     pid_t
     fork(void);

DESCRIPTION
     Fork() causes creation of a new process.  The new process (child process)
     is an exact copy of the calling process (parent process) except for the
     following:

           o   The child process has a unique process ID.

           o   The child process has a different parent process ID (i.e., the
               process ID of the parent process).

           o   The child process has its own copy of the parent's descriptors.
:
```

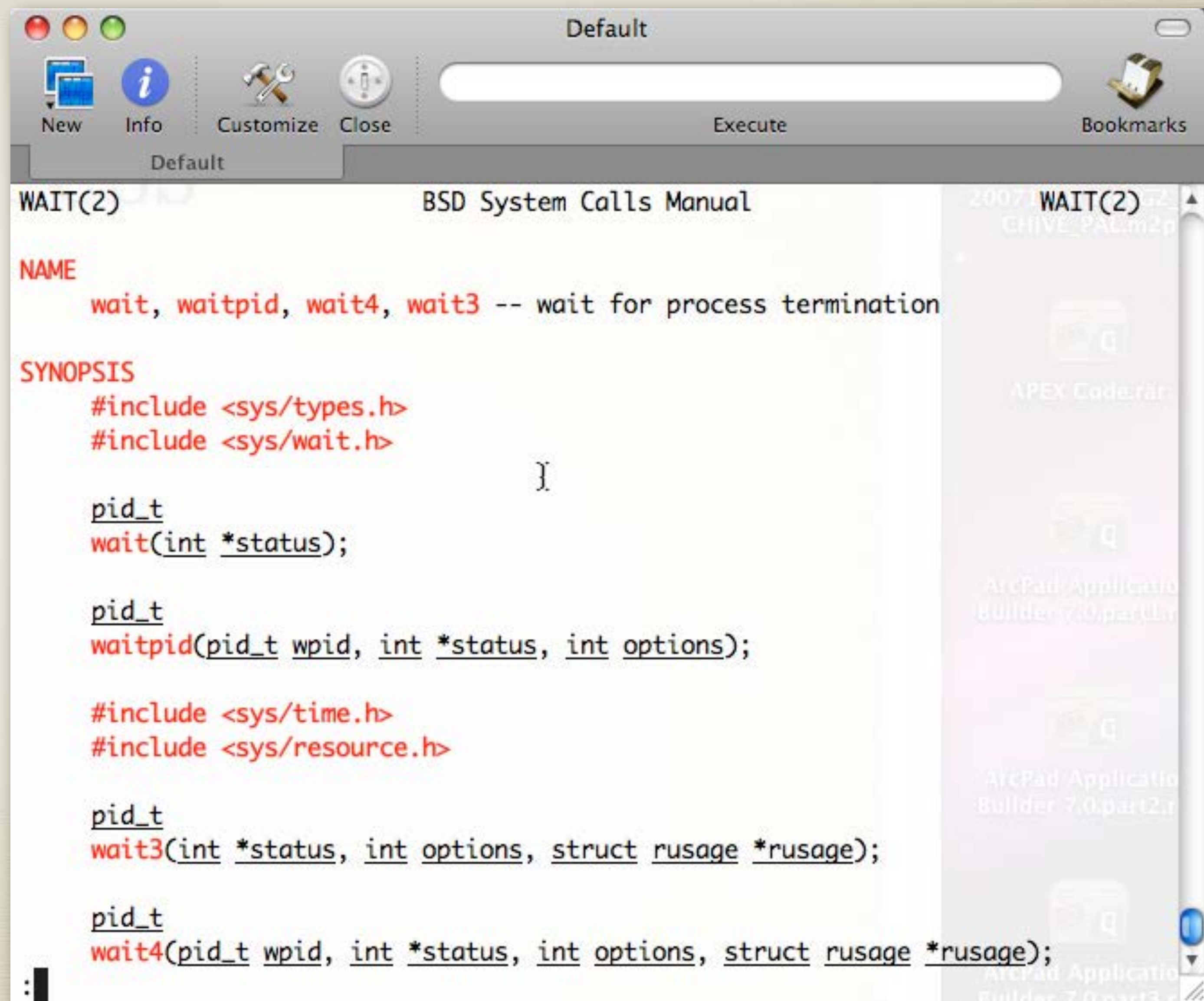
Process

* exec()



Process

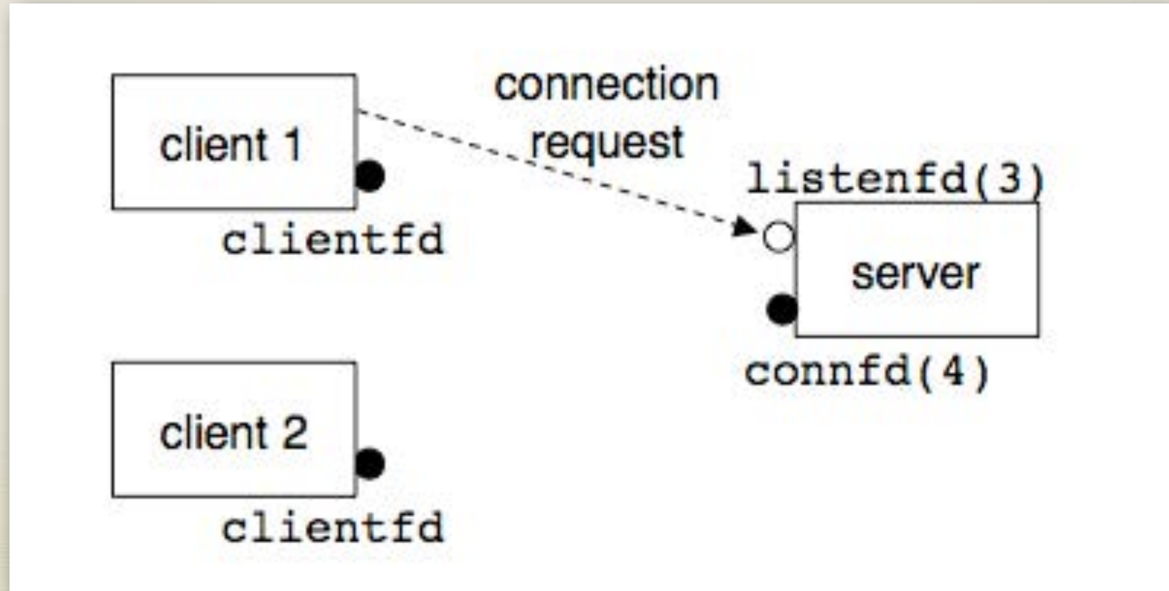
- waitpid( )



Process

* The easiest way for concurrency is PROCESS

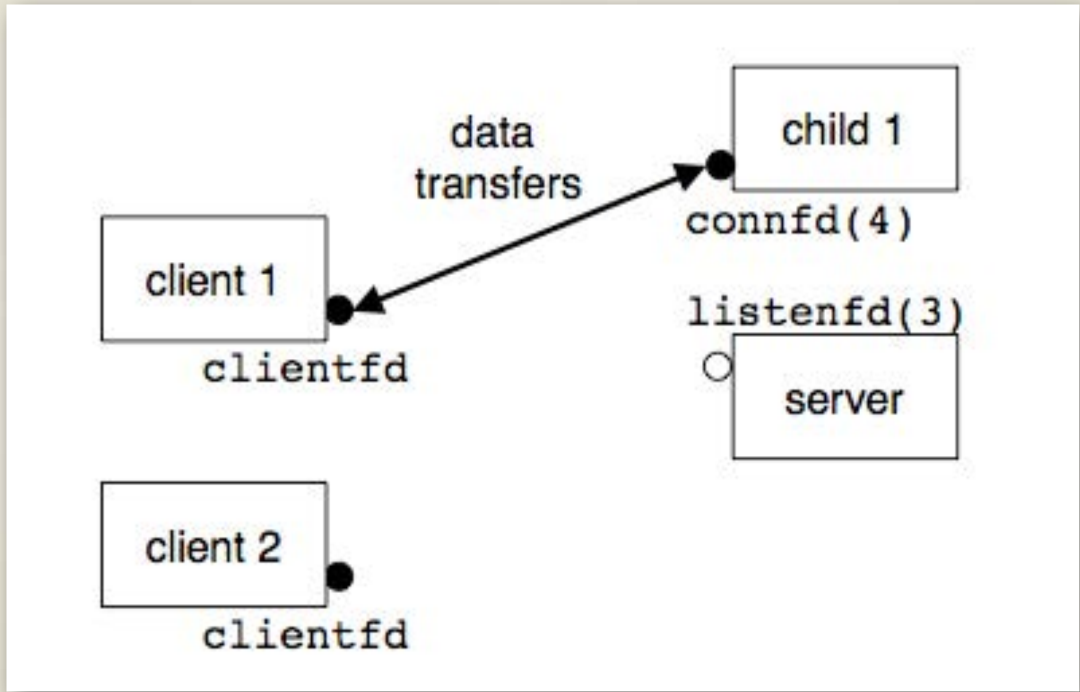
An example of process-based concurrent programming



STEP 1
Server accepts connection request from client

Process

An example of process-based concurrent programming

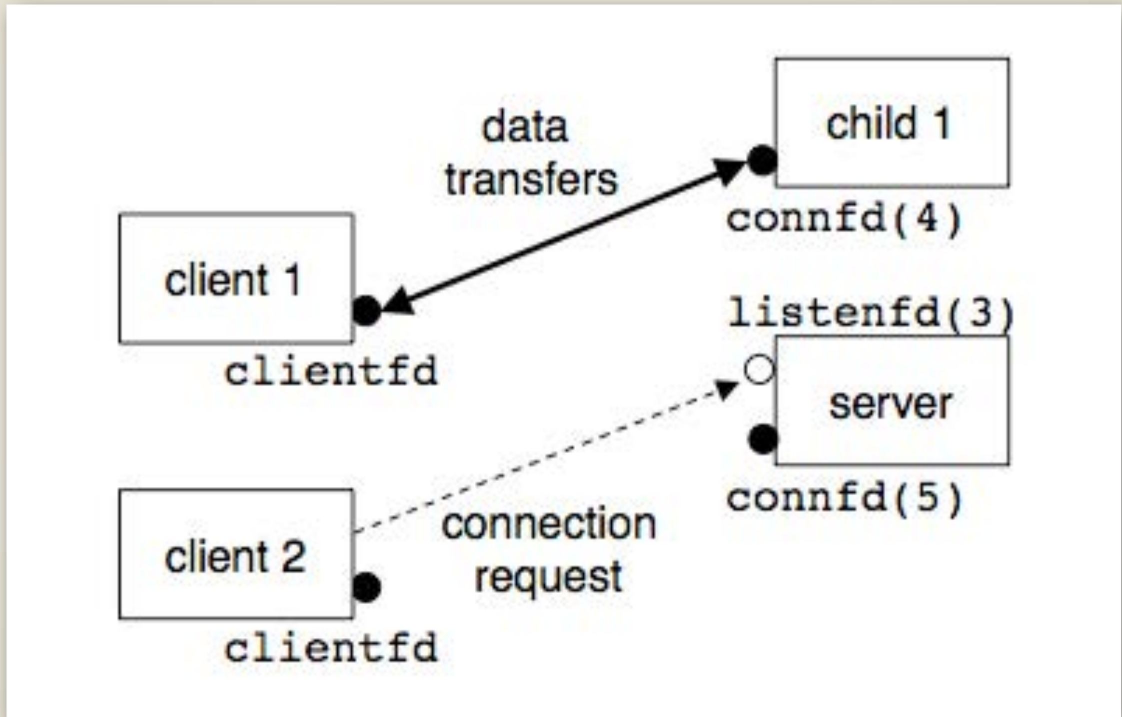


STEP 2

Server forks a child process to service the client

Process

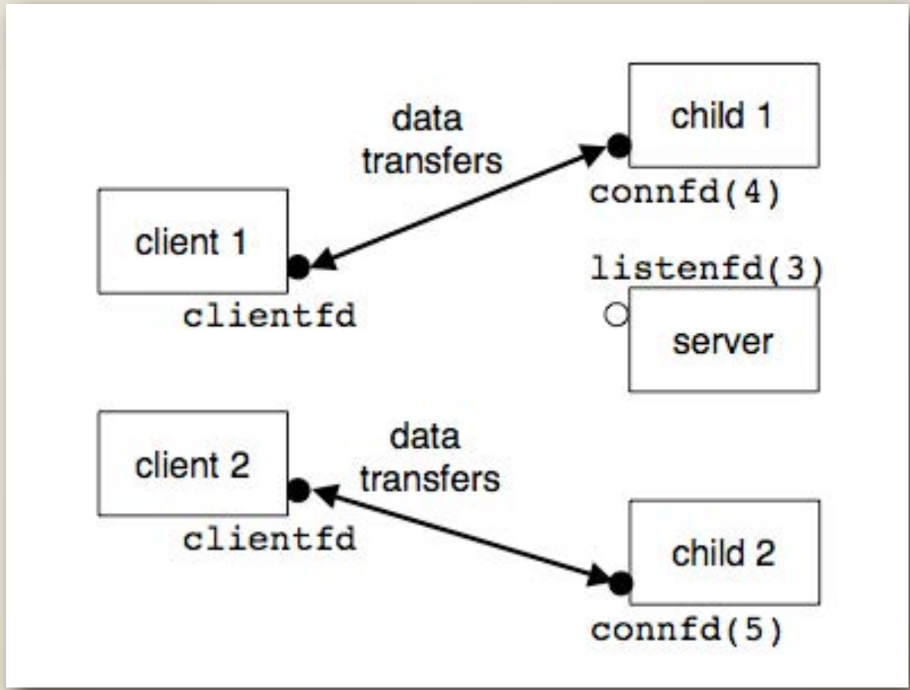
An example of process-based concurrent programming



STEP 3
Server accepts another connection request.

Process

An example of process-based concurrent programming



STEP 4
Server forks another child to service the new client.

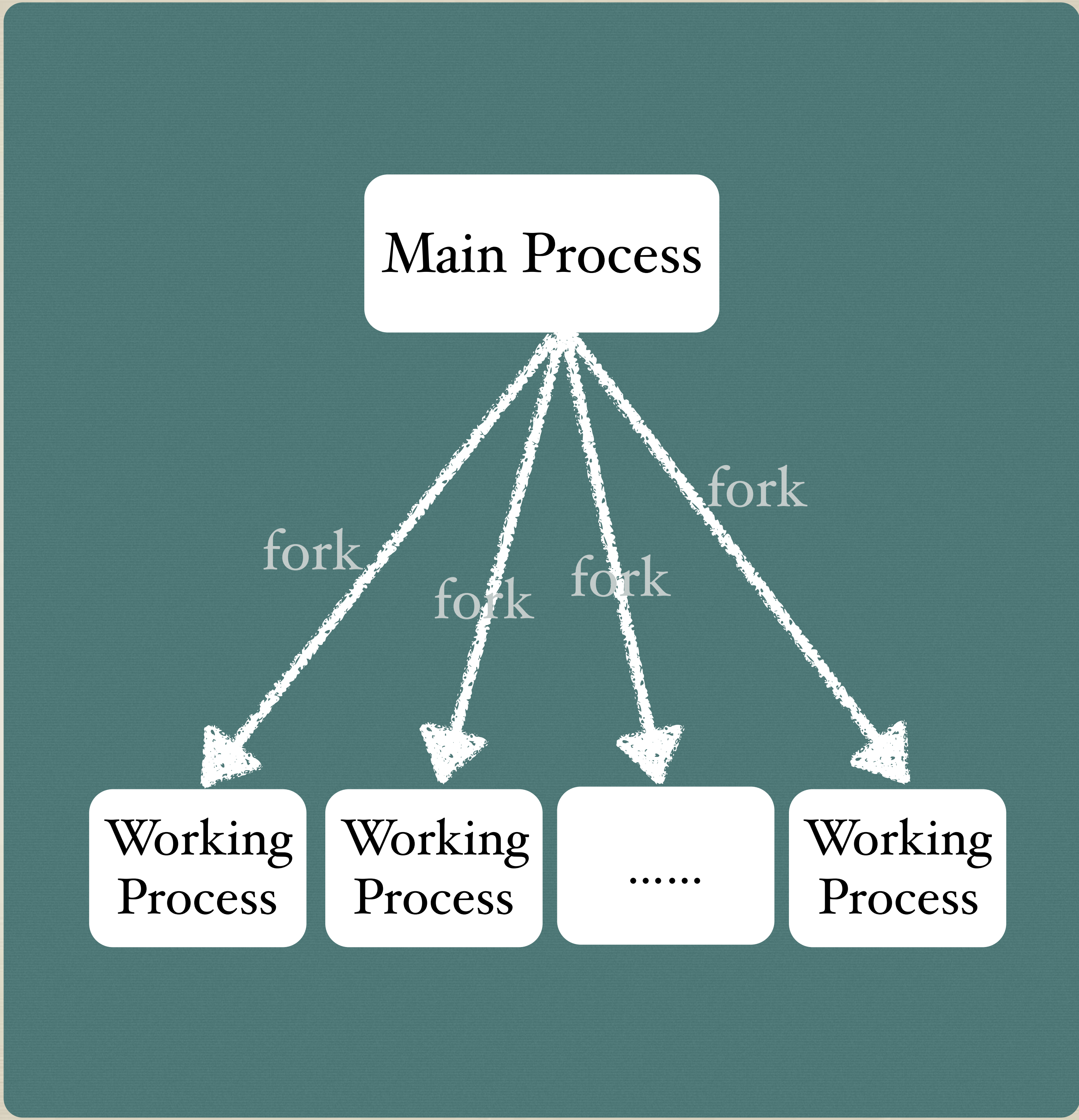
Process

Codes:

```
#include "csapp.h"

void echo(int connfd);

/* SIGCHLD signal handler */
void handler(int sig)
{
    pid_t pid;
    int stat;

    while ((pid = waitpid(-1, &stat, WNOHANG)) > 0)
        ;
    return;
}

int main(int argc, char **argv)
{
    int listenfd, connfd, port, clientlen;
    struct sockaddr_in clientaddr;

    if (argc != 2) {
        fprintf(stderr, "usage: %s <port>\n", argv[0]);
        exit(0);
    }
    port = atoi(argv[1]);

    Signal(SIGCHLD, handler);

    listenfd = open_listenfd(port);
    while (1) {
        clientlen = sizeof(clientaddr);
        connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
        if (Fork() == 0) {
            Close(listenfd); /* child closes its listening socket */
            echo(connfd);    /* child services client */
            Close(connfd);   /* child closes connection with client */
            exit(0);         /* child exits */
        }
        Close(connfd); /* parent closes connected socket (important!) */
    }
}
```

Process

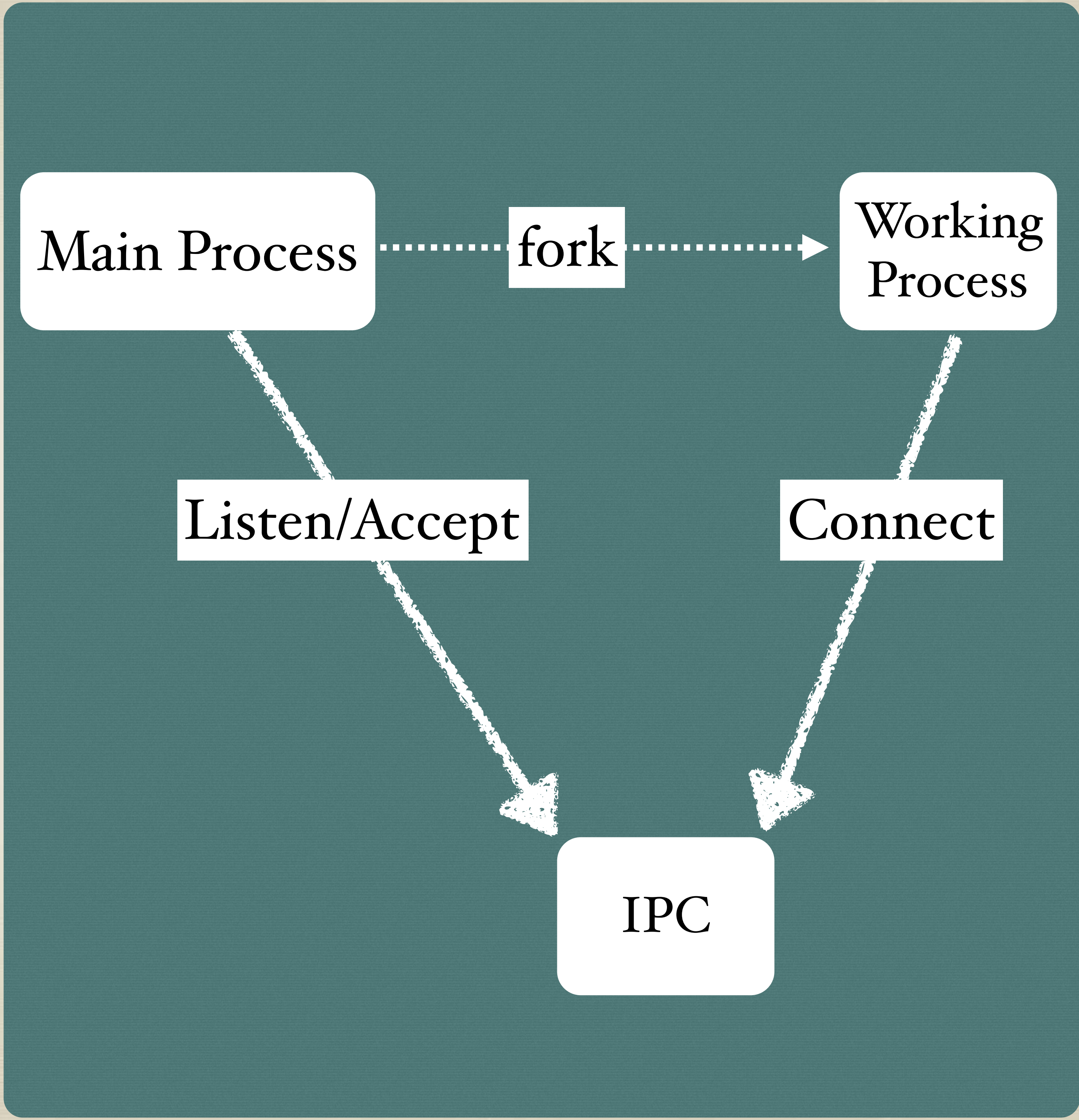
Model:



Process

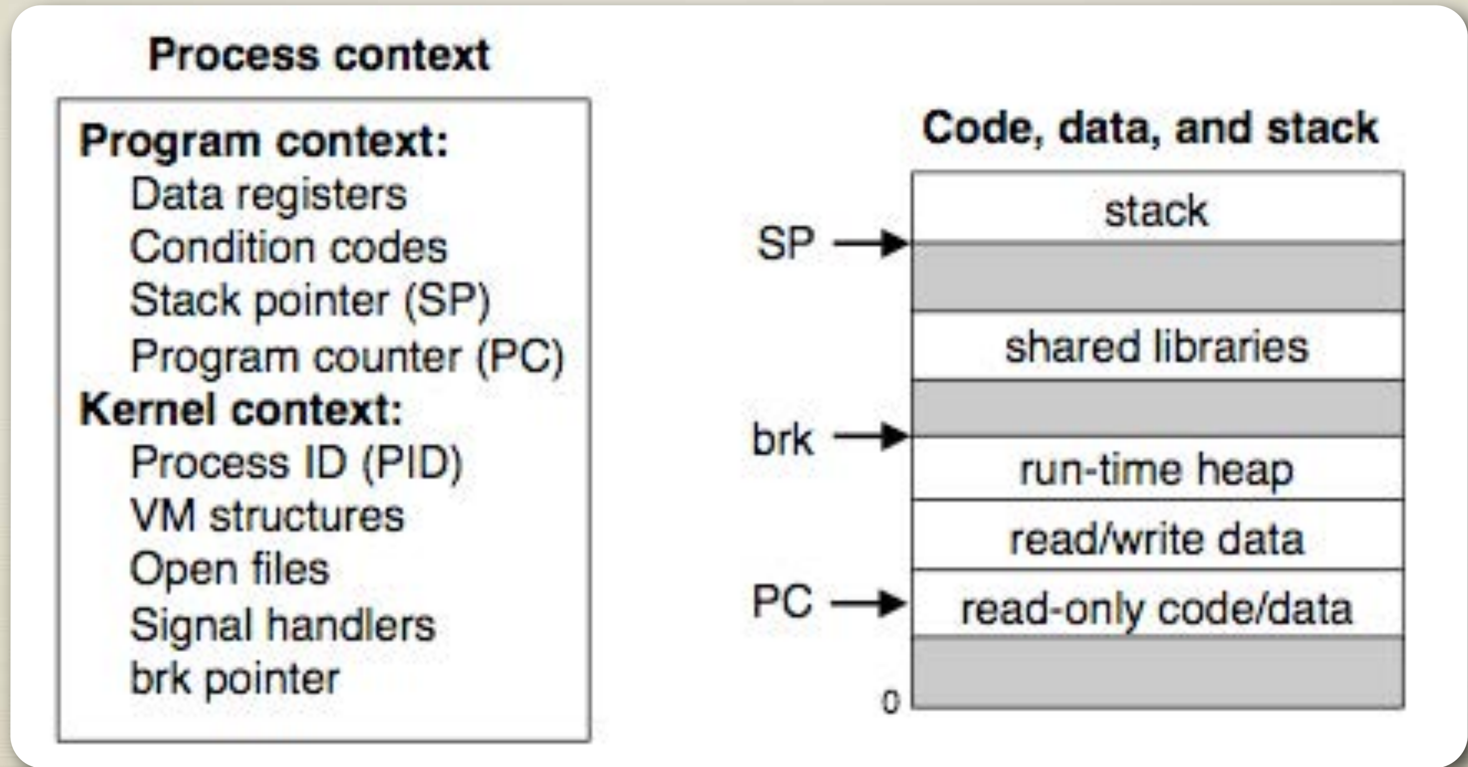
Communication

between process



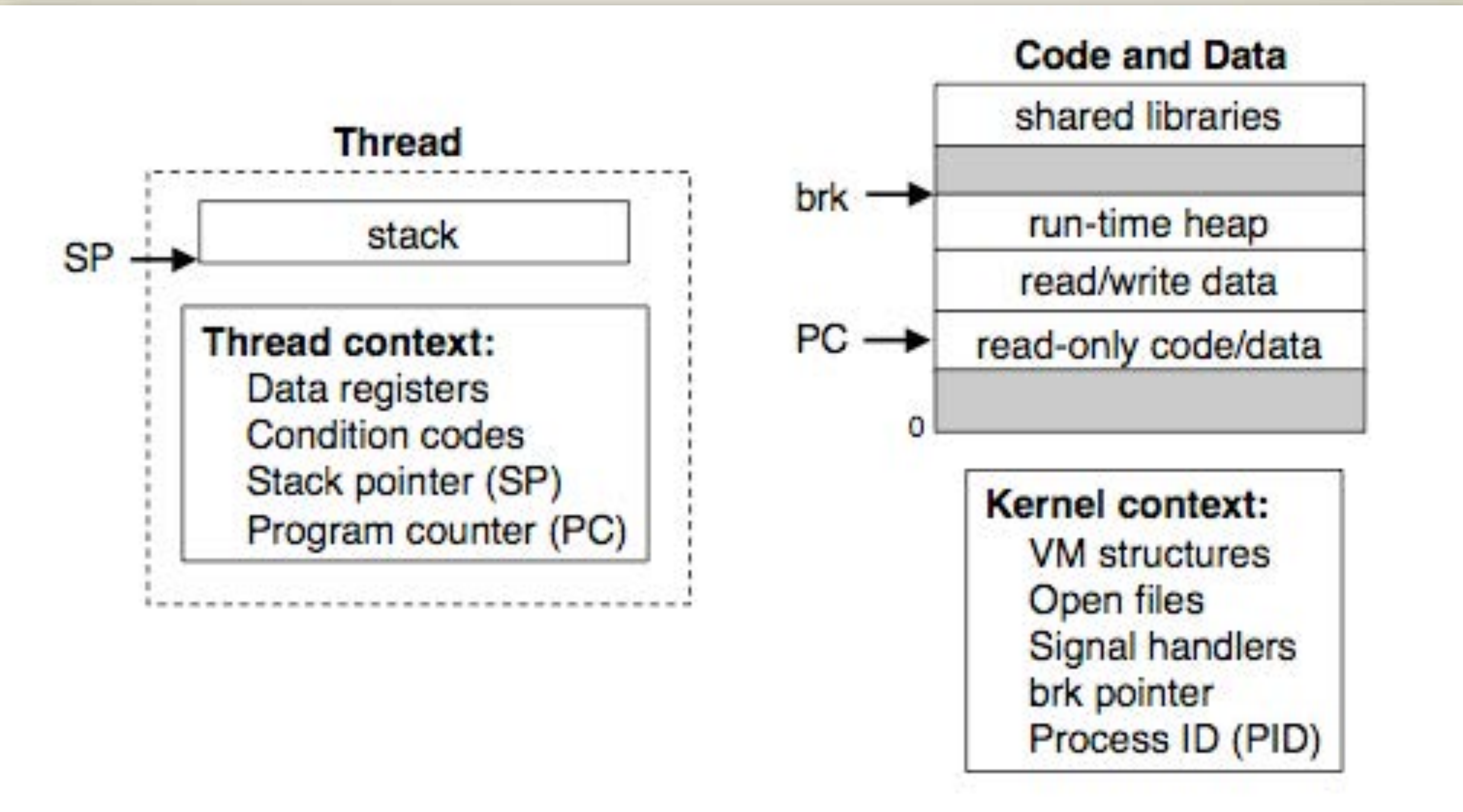
Threads

Traditional view of a process



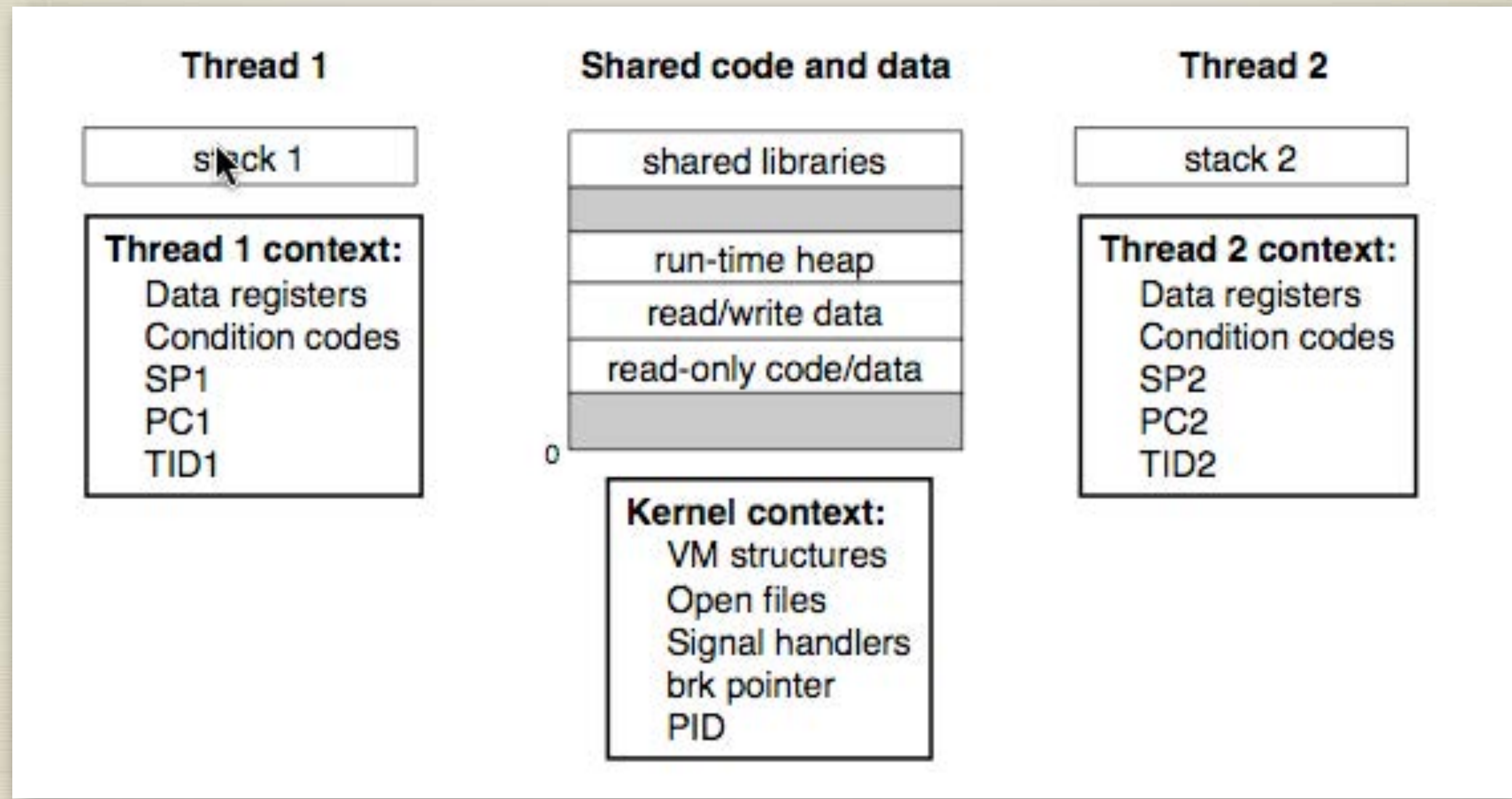
Threads

Alternative view of a process



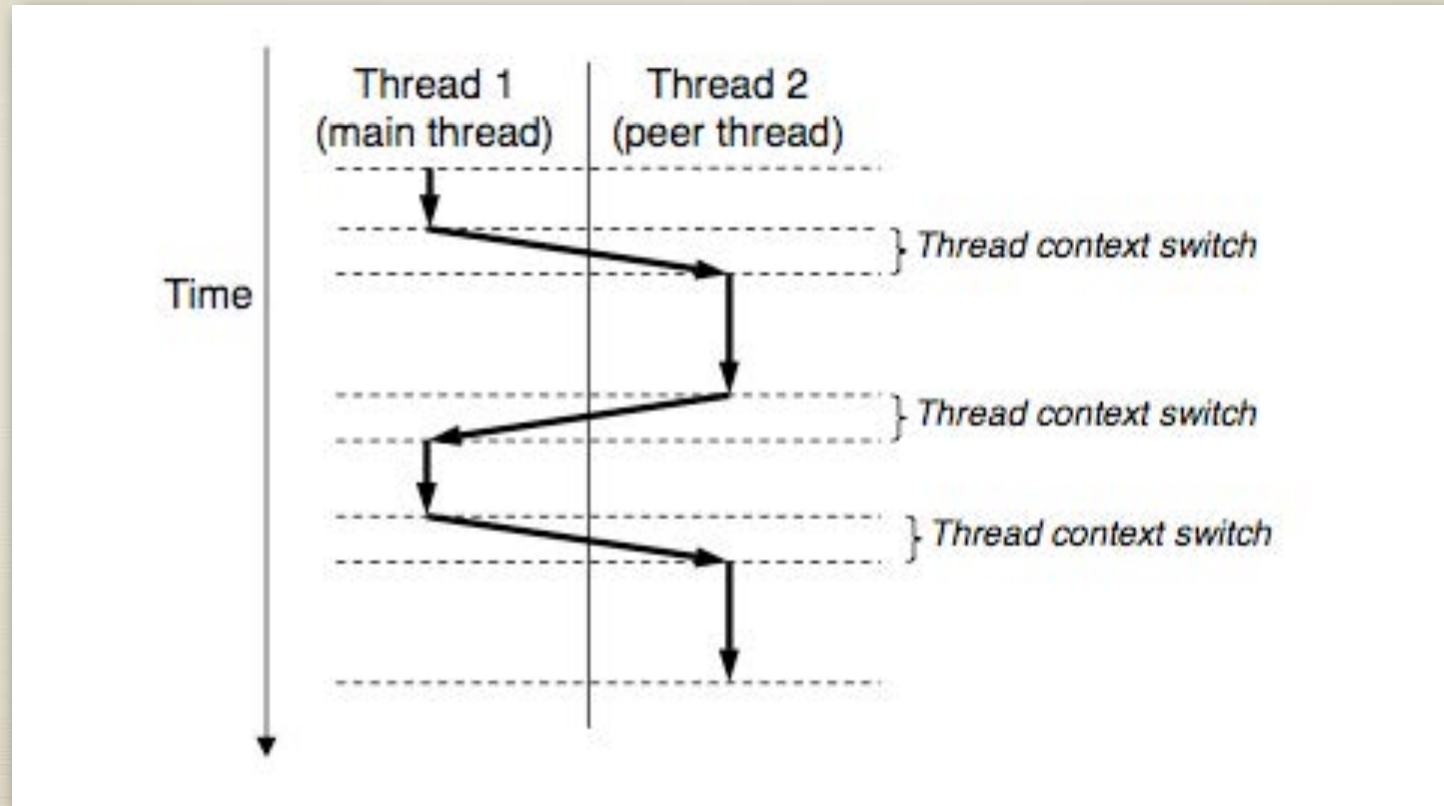
Threads

Associative multiple threads with a process



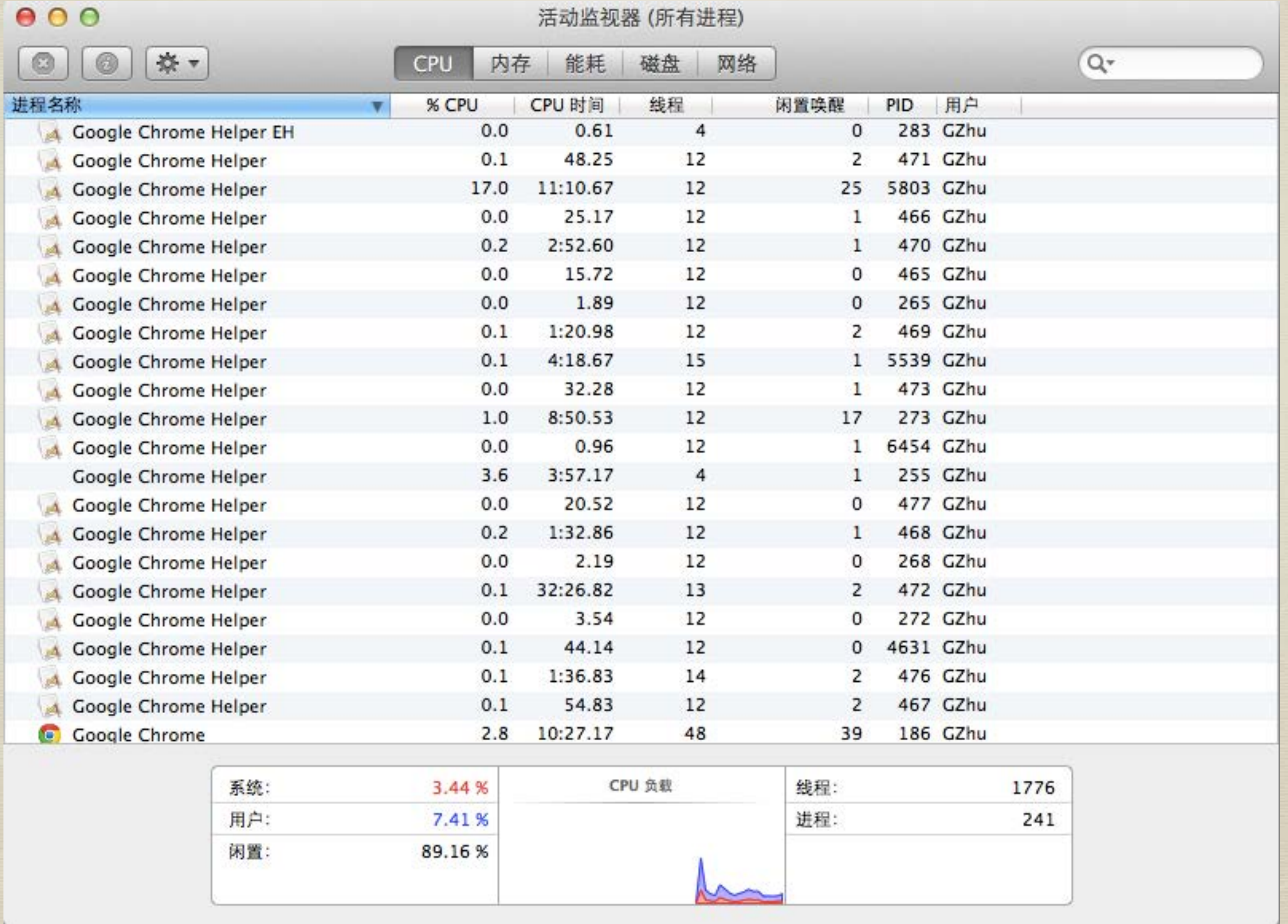
Threads

Concurrent thread execution



Threads

Concurrent thread execution



活动监视器 (所有进程)

CPU | 内存 | 能耗 | 磁盘 | 网络

进程名称	% CPU	CPU 时间	线程	闲置唤醒	PID	用户
Google Chrome Helper EH	0.0	0.61	4	0	283	GZhu
Google Chrome Helper	0.1	48.25	12	2	471	GZhu
Google Chrome Helper	17.0	11:10.67	12	25	5803	GZhu
Google Chrome Helper	0.0	25.17	12	1	466	GZhu
Google Chrome Helper	0.2	2:52.60	12	1	470	GZhu
Google Chrome Helper	0.0	15.72	12	0	465	GZhu
Google Chrome Helper	0.0	1.89	12	0	265	GZhu
Google Chrome Helper	0.1	1:20.98	12	2	469	GZhu
Google Chrome Helper	0.1	4:18.67	15	1	5539	GZhu
Google Chrome Helper	0.0	32.28	12	1	473	GZhu
Google Chrome Helper	1.0	8:50.53	12	17	273	GZhu
Google Chrome Helper	0.0	0.96	12	1	6454	GZhu
Google Chrome Helper	3.6	3:57.17	4	1	255	GZhu
Google Chrome Helper	0.0	20.52	12	0	477	GZhu
Google Chrome Helper	0.2	1:32.86	12	1	468	GZhu
Google Chrome Helper	0.0	2.19	12	0	268	GZhu
Google Chrome Helper	0.1	32:26.82	13	2	472	GZhu
Google Chrome Helper	0.0	3.54	12	0	272	GZhu
Google Chrome Helper	0.1	44.14	12	0	4631	GZhu
Google Chrome Helper	0.1	1:36.83	14	2	476	GZhu
Google Chrome Helper	0.1	54.83	12	2	467	GZhu
Google Chrome	2.8	10:27.17	48	39	186	GZhu

系统:	3.44 %
用户:	7.41 %
闲置:	89.16 %

CPU 负载

线程:	1776
进程:	241

Threads

Thread control

```
#include "csapp.h"

void *thread(void *vargp);

int main()
{
    pthread_t tid;

    Pthread_create(&tid, NULL, thread, NULL);
    Pthread_join(tid, NULL);
    exit(0);
}

/* thread routine */
void *thread(void *vargp)
{
    printf("Hello, world!\n");
    return NULL;
}
```

Threads

Thread control

```c
#include "csapp.h"

void *thread(void *vargp);

int main()
{
    pthread_t tid;

    Pthread_create(&tid, NULL, thread, NULL);
    Pthread_join(tid, NULL);
    exit(0);
}

/* thread routine */
void *thread(void *vargp)
{
    printf("Hello, world!\n");
    return NULL;
}
```

Could be a structured pointer

Main thread

Peer thread

Threads

Creating threads

```
#include <pthread.h>
typedef void *(func)(void *);

int pthread_create(pthread_t *tid, pthread_attr_t *attr, func *f, void *arg);
```

returns: 0 if OK, non-zero on error

The `pthread create` function creates a new thread and runs the *thread routine* `f` in the context of the new thread and with an input argument of `arg`. The `attr` argument can be used to change the default attributes of the newly created thread.

```
#include <pthread.h>

pthread_t pthread_self(void);
```

returns: thread ID of caller

When `pthread create` returns, argument `tid` contains the ID of the newly created thread. The new thread can determine its own thread ID by calling the `pthread self` function.

Threads

Terminating threads

```
#include <pthread.h>

int pthread_exit(void *thread_return);
```

returns: 0 if OK, non-zero on error

- The thread terminates *implicitly* when its top-level thread routine returns.
- The thread terminates *explicitly* by calling the `pthread exit` function, which returns a pointer to the return value `thread return`.

```
#include <pthread.h>

int pthread_cancel(pthread_t tid);
```

returns: 0 if OK, non-zero on error

- Some peer thread calls the Unix `exit` function, which terminates the process and all threads associated with the process.
- Another peer thread terminates the current thread by calling the `pthread cancel` function with the ID of the current thread.

Threads

Terminating threads

```
#include <pthread.h>

int pthread_join(pthread_t tid, void **thread_return);
```

returns: 0 if OK, non-zero on error

The `pthread join` function blocks until thread `tid` terminates, assigns the `(void *)` pointer returned by the thread routine to the location pointed to by `thread return`, and then *reaps* any memory resources held by the terminated thread.

Threads

Detaching threads

```c
#include <pthread.h>

int pthread_detach(pthread_t tid);
```

returns: 0 if OK, non-zero on error

The `pthread detach` function detaches the joinable thread `tid`. Threads can detach themselves by calling `pthread detach` with an argument of `pthread self()`.

- At any point in time, a thread is *joinable* or *detached*.
- A joinable thread can be reaped and killed by other threads. Its memory resources (such as the stack) are not freed until it is reaped by another thread.
- In contrast, a detached thread cannot be reaped or killed by other threads. Its memory resources are freed automatically by the system when it terminates.

For example, a high-performance Web server

Threads

Debugging

A. The thread is supposed to sleep for one second and then print a string. However, when we run it, nothing prints. Why?

B. You can fix this bug by replacing the `exit` function in line 9 with one of two different Pthreads function calls. Which ones?

```
1 #include "csapp.h"
2 void *thread(void *vargp);
3
4 int main()
5 {
6     pthread_t tid;
7
8     Pthread_create(&tid, NULL, thread, NULL);
9     exit(0);
10 }
11
12 /* thread routine */
13 void *thread(void *vargp)
14 {
15     Sleep(1);
16     printf("Hello, world!\n");
17     return NULL;
18 }
```

A. The problem is that the main thread calls `exit` without waiting for the peer thread to terminate. The `exit` call terminates the entire process, including any threads that happen to be running. So the peer thread is being killed before it has a chance to print its output string.

B. We can fix the bug by replacing the `exit` function with either `pthread exit`, which waits for outstanding threads to terminate before it terminates the process, or `pthread join` which explicitly reaps the peer thread.

Threads

Shared variables in threaded program

```c
#include "csapp.h"
#define N 2

char **ptr;  /* global variable */

void *thread(void *vargp);

int main()
{
    int i;
    pthread_t tid;
    char *msgs[N] = {
        "Hello from foo",
        "Hello from bar"
    };

    ptr = msgs;

    for (i = 0; i < N; i++)
        Pthread_create(&tid, NULL, thread, (void *)i);
    Pthread_exit(NULL);
}

void *thread(void *vargp)
{
    int myid = (int)vargp;
    static int cnt = 0;

    printf("[%d]: %s (cnt=%d)\n", myid, ptr[myid], ++cnt);
}
```

```
[0]: Hello from foo (cnt=1)
[1]: Hello from bar (cnt=2)
```

Threads

Shared variables in threaded program

QUESTION

(1) What is the underlying memory model for threads?

(2) Given this model, how are instances of the variable mapped to memory?

(3) And finally, how many threads reference each of these instances?

Threads

Threads memory model

- A pool of concurrent threads runs in the context of a process. Each thread has its own separate thread context, which includes a thread ID, stack, stack pointer, program counter, condition codes, and general purpose register values.
- Each thread shares the rest of the process context with the other threads.
- In an operational sense, it is impossible for one thread to read or write the register values of another thread. On the other hand, any thread can access any location in the shared virtual memory. Thus, registers are never shared, while virtual memory is always shared.
- The memory model for the separate thread stacks is not as clean. These stacks are contained in the stack area of the virtual address space, and are *usually* accessed independently by their respective threads. We say *usually* rather than *always*, because different thread stacks are not protected from other threads.

Threads

Th

Global variables. A *global variable* is any variable declared outside of a function. For example, the global `ptr` variable in line 4 has one run-time instance in the read/write area of virtual memory.

Local automatic variables. A *local automatic variable* is one that is declared inside a function without the `static` attribute. For example, there is one instance of the local variable `tid`, and it resides on the stack of the main thread.

- `tid.m`
- `myid.p0`
- `myid.p1`

Local static variables. A *local static variable* is one that is declared inside a function with the `static` attribute. For example, `cnt` in line 27, at runtime there is only one instance of `cnt` residing in the read/write area of virtual memory. Each peer thread reads and writes this instance.

```
1 #include "csapp.h"
2 #define N 2
3
4 char **ptr;  /* global variable */
5
6 void *thread(void *vargp);
7
8 int main()
9 {
10     int i;
11     pthread_t tid;
12     char *msgs[N] = {
13         "Hello from foo",
14         "Hello from bar"
15     };
16
17     ptr = msgs;
18
19     for (i = 0; i < N; i++)
20         Pthread_create(&tid, NULL, thread, (void *)i);
21     Pthread_exit(NULL);
22 }
23
24 void *thread(void *vargp)
25 {
26     int myid = (int)vargp;
27     static int cnt = 0;
28
29     printf("[%d]: %s (cnt=%d)\n", myid, ptr[myid], ++cnt);
30 }
```

Threads

Thr

Variable instance	Referenced by main thread	Referenced by peer thread 0	Referenced by peer thread 1
ptr	Y	Y	Y
cnt	N	Y	Y
i.m	Y	N	N
msg.m	Y	Y	Y
myid.p0	N	Y	N
myid.p1	N	N	Y

```
#include "csapp.h"
#define N 2

char **ptr;  /* global variable */

void *thread(void *vargp);

int main()
{
    int i;
    pthread_t tid;
    char *msgs[N] = {
        "Hello from foo",
        "Hello from bar"
    };

    ptr = msgs;

    for (i = 0; i < N; i++)
        Pthread_create(&tid, NULL, thread, (void *)i);
    Pthread_exit(NULL);
}

void *thread(void *vargp)
{
    int myid = (int)vargp;
    static int cnt = 0;

    printf("[%d]: %s (cnt=%d)\n", myid, ptr[myid], ++cnt);
}
```

Threads

Synchronizing threads with semaphores

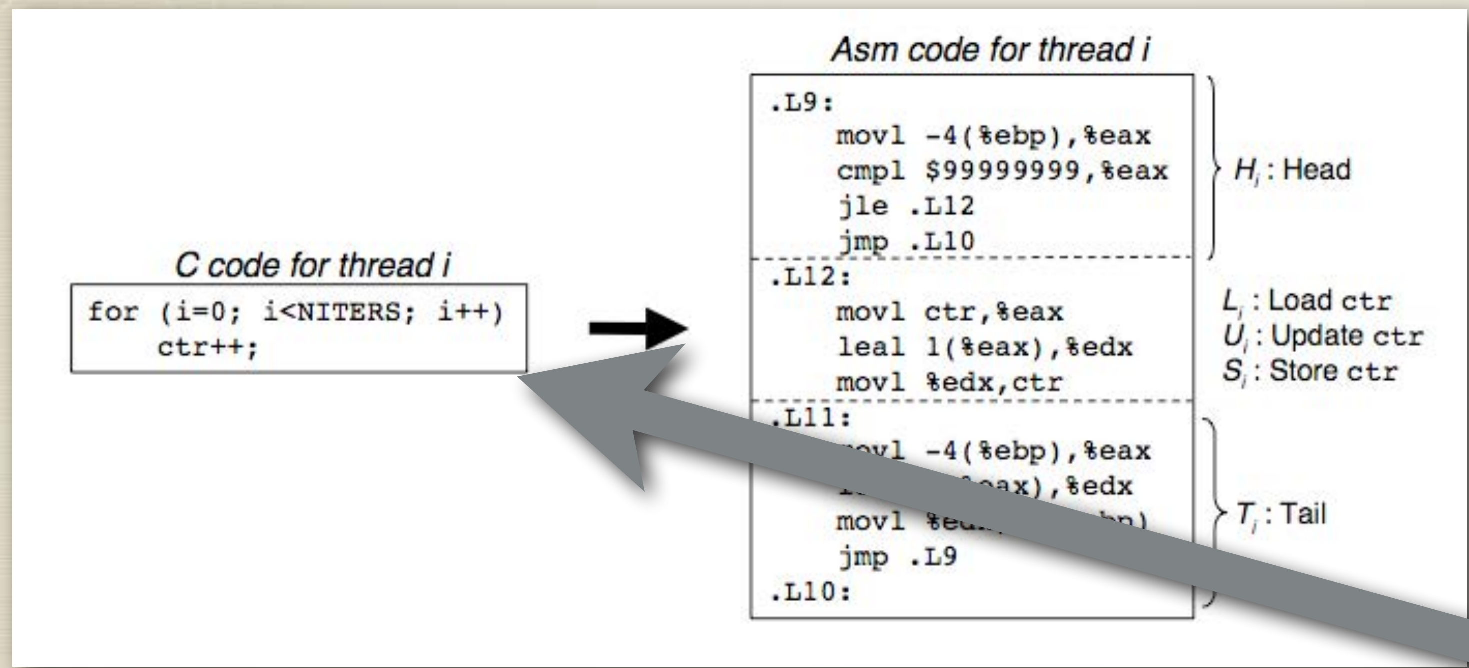
```
unix> ./badcnt
BOOM!  cnt=198841183

unix> ./badcnt
BOOM!  cnt=198261802

unix> ./badcnt
BOOM!  cnt=198269768
```

?

```c
#include "csapp.h"

#define NITERS 100000000

void *count(void *arg);

/* shared variable */
unsigned int cnt = 0;

int main()
{
    pthread_t tid1, tid2;

    Pthread_create(&tid1, NULL, count, NULL);
    Pthread_create(&tid2, NULL, count, NULL);

    Pthread_join(tid1, NULL);
    Pthread_join(tid2, NULL);

    if (cnt != (unsigned)NITERS*2)
        printf("BOOM! cnt=%d\n", cnt);
    else
        printf("OK cnt=%d\n", cnt);
    exit(0);
}

/* thread routine */
void *count(void *arg)
{
    int i;

    for (i=0; i<NITERS; i++)
        cnt++;
    return NULL;
}
```

Threads

Synchronizing threads with semaphores




```c
#include "csapp.h"

#define NITERS 100000000

void *count(void *arg);

/* shared variable */
unsigned int cnt = 0;

int main()
{
    pthread_t tid1, tid2;

    Pthread_create(&tid1, NULL, count, NULL);
    Pthread_create(&tid2, NULL, count, NULL);

    Pthread_join(tid1, NULL);
    Pthread_join(tid2, NULL);

    if (cnt != (unsigned)NITERS*2)
        printf("BOOM! cnt=%d\n", cnt);
    else
        printf("OK cnt=%d\n", cnt);
    exit(0);
}

/* thread routine */
void *count(void *arg)
{
    int i;

    for (i=0; i<NITERS; i++)
        cnt++;
    return NULL;
}
```

- H_i: The block of instructions at the head of the loop.
- L_i: The instruction that loads the shared variable `cnt` into register `%eax`$_i$, where `%eax`$_i$ denotes the value of register `%eax` in thread i.
- U_i: The instruction that updates (increments) `%eax`$_i$.
- S_i: The instruction that stores the updated value of `%eax`$_i$ back to the shared variable `cnt`.
- T_i: The block of instructions at the tail of the loop.

Threads

Synchronizing threads with semaphores

C code for thread i

```
for (i=0; i<NITERS; i++)
    ctr++;
```

Asm code for thread i

```
.L9:
    movl -4(%ebp),%eax
    cmpl $99999999,%eax
    jle .L12
    jmp .L10
.L12:
    movl ctr,%eax
    leal 1(%eax),%edx
    movl %edx,ctr
.L11:
    movl -4(%ebp),%eax
    leal 1(%eax),%edx
    movl %edx,-4(%ebp)
    jmp .L9
.L10:
```

H_i : Head

L_i : Load `ctr`
U_i : Update `ctr`
S_i : Store `ctr`

T_i : Tail

Step	Thread	Instr	`%eax`$_1$	`%eax`$_2$	`cnt`
1	1	H_1	–	–	0
2	1	L_1	0	–	0
3	1	U_1	1	–	0
4	1	S_1	1	–	1
5	2	H_2	–	–	1
6	2	L_2	–	1	1
7	2	U_2	–	2	1
8	2	S_2	–	2	2
9	2	T_2	–	2	2
10	1	T_1	1	–	2

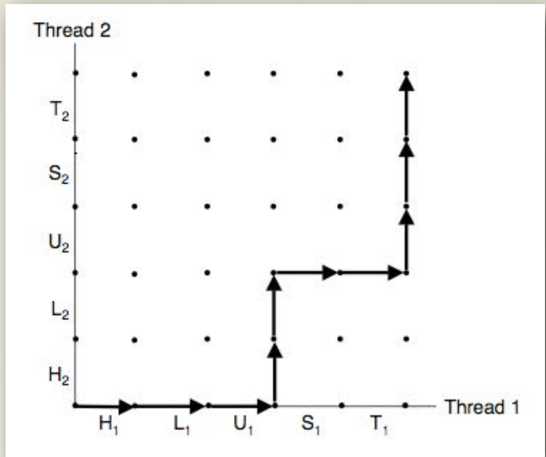
(a) Correct ordering

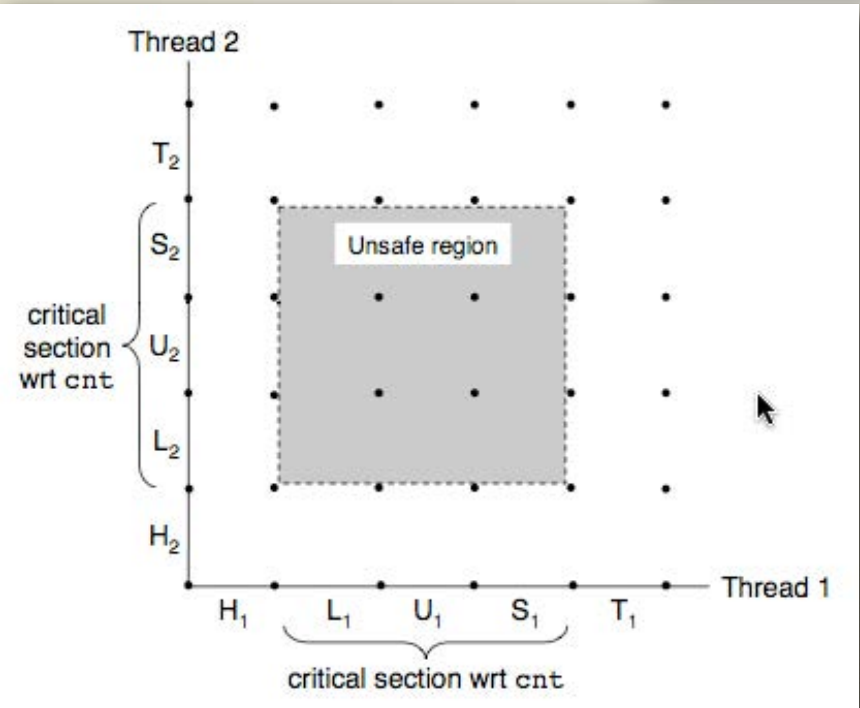
Step	Thread	Instr	`%eax`$_1$	`%eax`$_2$	`cnt`
1	1	H_1	–	–	0
2	1	L_1	0	–	0
3	1	U_1	1	–	0
4	2	H_2	–	–	0
5	2	L_2	–	0	0
6	1	S_1	1	–	1
7	1	T_1	1	–	1
8	2	U_2	–	1	1
9	2	S_2	–	1	1
10	2	T_2	–	1	1

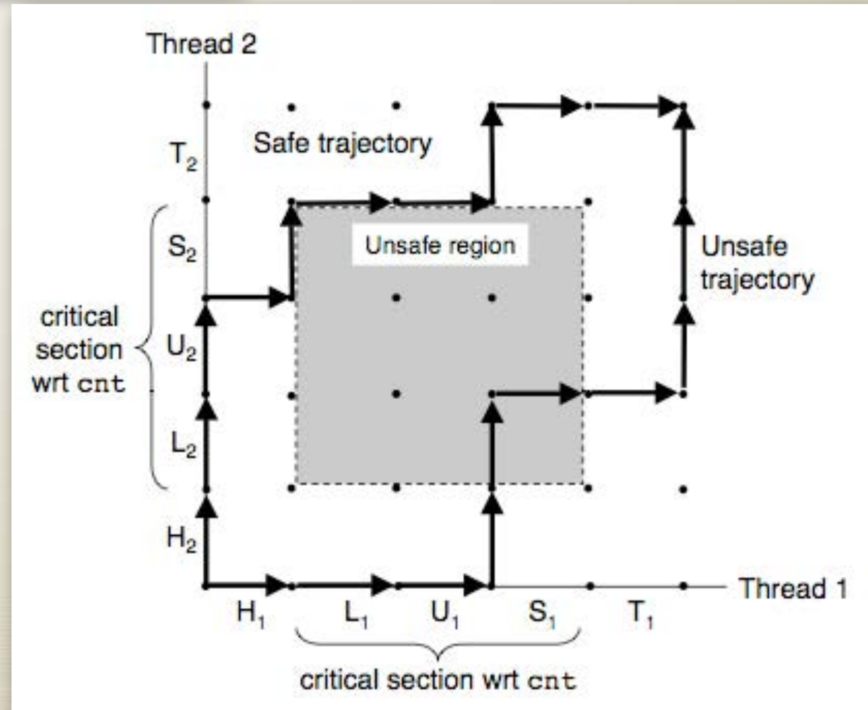
(b) Incorrect ordering

Threads

Progress Graphs







Threads

Synchronizing threads!

A *semaphore*, *s*, is a global variable with a nonnegative integer value that can only be manipulated by two special operations, called *P* and *V*:

P(s): while (s <= 0) ; s--;
V(s): s++;

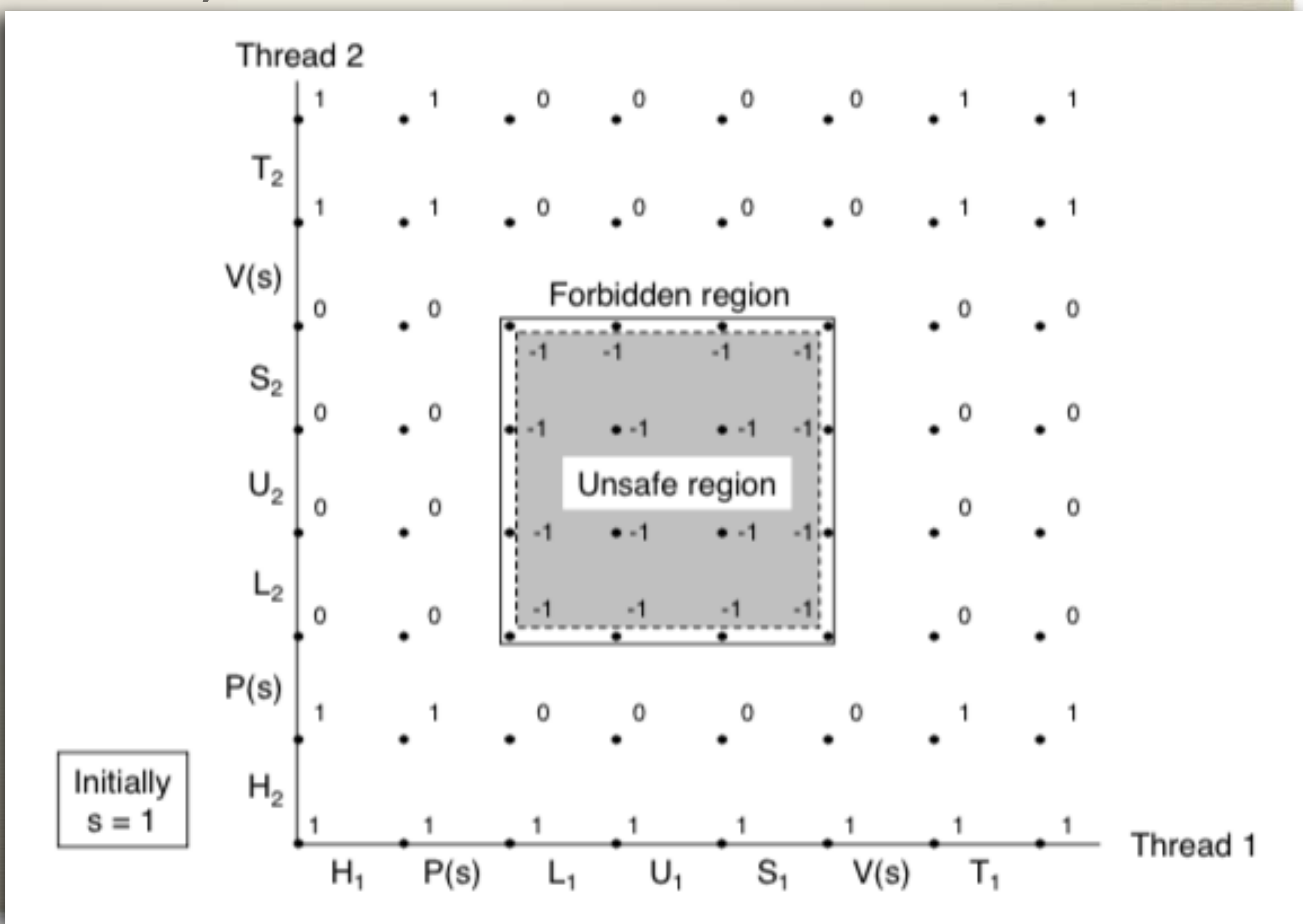
The definitions of and ensure that a running program can never enter a state where a properly initialized semaphore has a negative value. This property, known as the *semaphore invariant*, provides a powerful tool for controlling the trajectories of concurrent programs so that they avoid unsafe regions.

Threads

Synchronizing threads!

P(s): while (s <= 0) ; s--;

V(s): s++;



Threads

Synchronizing threads with

Mutex & Condition variables

Mutex variables
Condition variables
Barrier synchronization
Timeout waiting

Threads

Thread-safe and reentrant functions

Failing to protect shared variables

```c
#include "csapp.h"

#define NITERS 100000000

void *count(void *arg);

/* shared variable */
unsigned int cnt = 0;

int main()
{
    pthread_t tid1, tid2;

    Pthread_create(&tid1, NULL, count, NULL);
    Pthread_create(&tid2, NULL, count, NULL);

    Pthread_join(tid1, NULL);
    Pthread_join(tid2, NULL);

    if (cnt != (unsigned)NITERS*2)
        printf("BOOM! cnt=%d\n", cnt);
    else
        printf("OK cnt=%d\n", cnt);
    exit(0);
}

/* thread routine */
void *count(void *arg)
{
    int i;

    for (i=0; i<NITERS; i++)
        cnt++;
    return NULL;
}
```

Threads

Thread-safe and reentrant functions

Relying on state across multiple function invocations

```
unsigned int next = 1;

/* rand - return pseudo-random integer on 0..32767 */
int rand(void)
{
    next = next*1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}

/* srand - set seed for rand() */
void srand(unsigned int seed)
{
    next = seed;
}
```

Threads

Thread-safe and reentrant functions

and, MORE...

Returning a pointer to a static variable

Calling thread-unsafe functions

Threads

Thread-safe library functions

Thread-unsafe function	Thread-unsafe class	Unix thread-safe version
`asctime`	3	`asctime_r`
`ctime`	3	`ctime_r`
`gethostbyaddr`	3	`gethostbyaddr_r`
`gethostbyname`	3	`gethostbyname_r`
`inet_ntoa`	3	(none)
`localtime`	3	`localtime_r`
`rand`	2	`rand_r`