

# Threaded Architectures for Internet Servers

RBG-Seminar WS 2008/2009

09.12.2008


Holger Kälberer

# Agenda

- (1) Overview: (P)threads, Internet-Server
- (2) Threaded Serve-Strategies
- (3) Subsequent Network Communication

# 1. Overview

# (1) Threads

- Thread = flow of instructions
  - multi -threading, -tasking, -processor
  - sequential, concurrent and parallel programming
  - priorities, scheduling
  - library-/user- (N:1) vs. kernel-thread model (1:1)
    - userlevel
    - fast context switches
    - high portability
    - GNU Pth, FSU Pthreads
    - hybrid models (M:N)
    - green threads
    - kernel support
    - true concurrency
    - limited portability
    - NPTL, Solaris  $\geq 9$
    - hybrid models (M:N; NetBSD)
- 

# Process vs. (native) Thread

- running instance of a program
- = address space + flow of control (thread/s)
- part of a process
- = stack + process state + instruction ptr. (+ thread-spec. data)
- shared heap, bss + data seg. + unix attributes

## ... cont.

- heavyweight:
  - context switches
  - creation: fork()
- communication + shared resources: complex and expensive (shared memory, pipes, ...)
- lightweight (LWP)
  - creation: pthread\_create()
- ... simple and cheaper but dangerous ...

# Thread programming pitfalls

- race conditions
- deadlocks
- non thread-safe functions (libc, glib, talloc, ...)
- lock contention

# Pthreads: background

- POSIX 1003.1c-1995: Pthreads
- LINUX/glibc implementation: Native POSIX Threading Library (NPTL).
  - replaces LinuxThreads (2.0 – 2.4)
  - kernel-support with NPTL (2.6, glibc 2.3.2)
  - 1:1 model
  - futex(2)
  - no manager thread

# <pthread.h>

## MANAGEMENT

```
int pthread_create(pthread_t *, const pthread_attr_t *,
void (*)(void *), void *);
void pthread_exit(void *);
int pthread_cancel(pthread_t);
void pthread_cleanup_push(void*, void *);
void pthread_cleanup_pop(int);
int pthread_join(pthread_t thread, void **value_ptr);

int pthread_attr_init(pthread_attr_t *);
int pthread_attr_setstacksize(pthread_attr_t *attr, size_t
stacksize);
int pthread_attr_getstacksize(const pthread_attr_t *attr,
size_t *stacksize);
```

## TLS

```
int pthread_key_create(pthread_key_t *,
void (*)(void *));
void *pthread_getspecific(pthread_key_t);
int pthread_setspecific(pthread_key_t key,
const void *value);
int pthread_key_delete(pthread_key_t);
```

## LOCKING

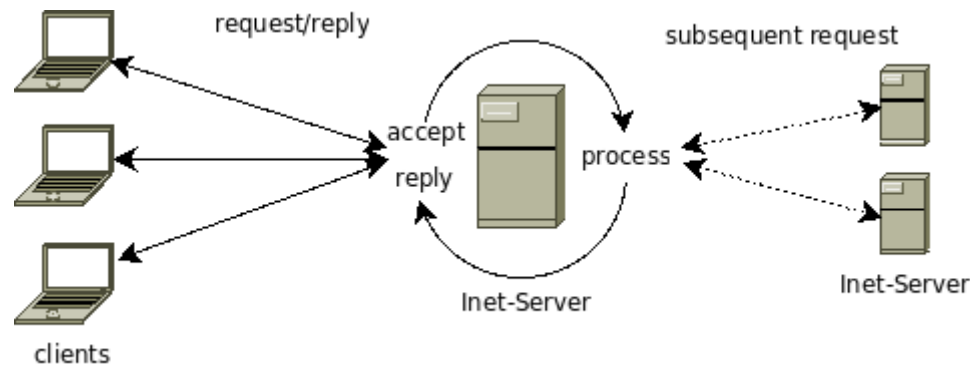
```
int pthread_mutex_init(pthread_mutex_t *, const
pthread_mutexattr_t *);
int pthread_mutex_lock(pthread_mutex_t *);
int pthread_mutex_unlock(pthread_mutex_t *);
int pthread_mutexattr_init(pthread_mutexattr_t *);

int pthread_rwlock_init(pthread_rwlock_t *,
const pthread_rwlockattr_t *);
int pthread_rwlock_rdlock(pthread_rwlock_t *);
int pthread_rwlock_wrlock(pthread_rwlock_t *);
int pthread_rwlock_unlock(pthread_rwlock_t *);
```

## CONDITIONS

```
int pthread_cond_init(pthread_cond_t *, const
pthread_condattr_t *);
int pthread_cond_wait(pthread_cond_t *,
pthread_mutex_t *);
int pthread_cond_broadcast(pthread_cond_t *);
int pthread_cond_signal(pthread_cond_t *);
int pthread_cond_timedwait(pthread_cond_t *,
pthread_mutex_t *, const struct timespec *);
```

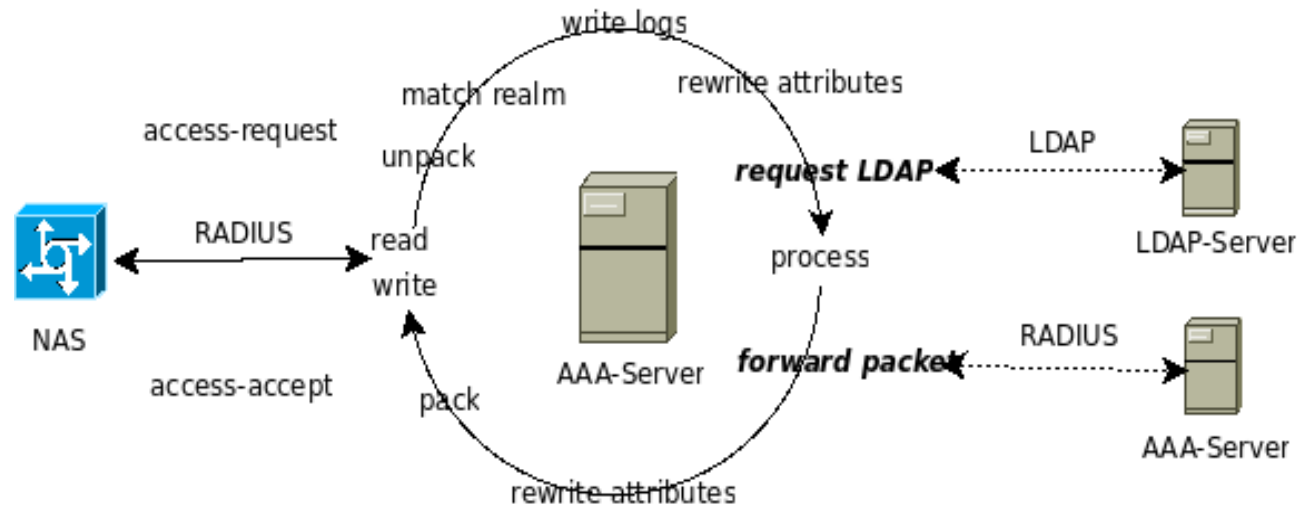
# Overview (2): Internet Server



1. connection management (demultiplexing/event-handling)
2. request processing
3. reply

# Examples

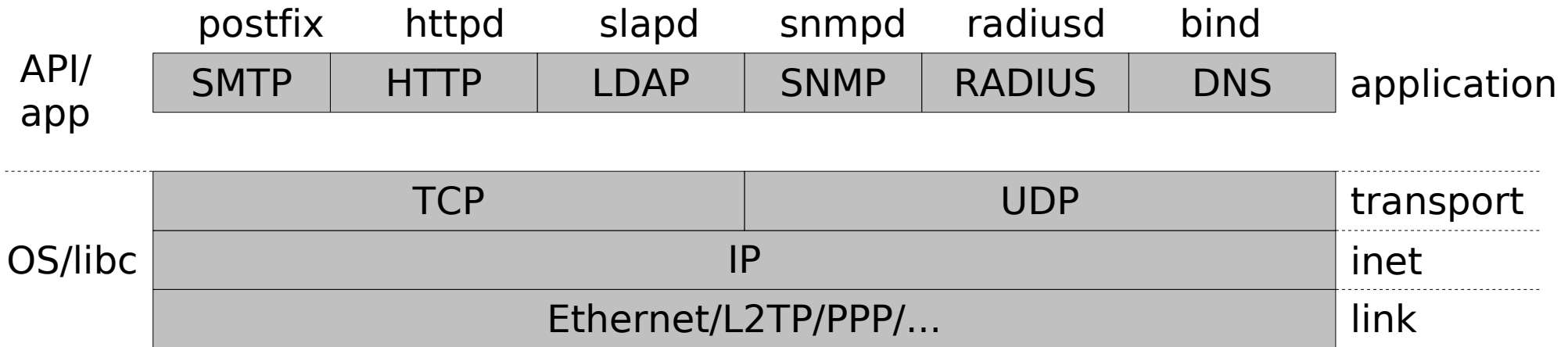
- RADIUS



- Webserver

- memcached

# Network Communication



- low-level system-calls:
- higher-level APIs:

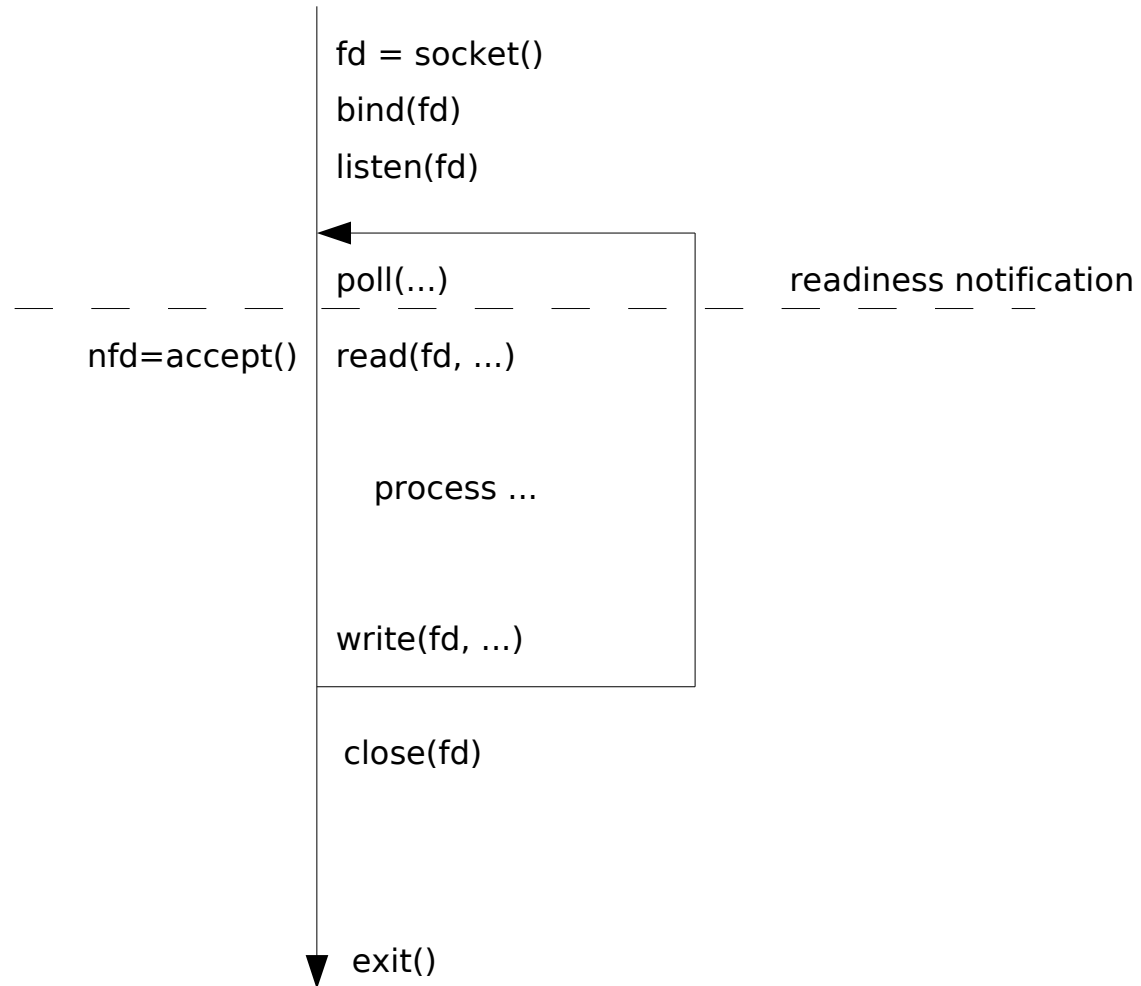
```
int ldap_search_ext(...)
int mysql_query(MYSQL *mysql, const char
               *query);
...
```

```
socket(2)
bind(2)
listen(2)
select(2)/poll(2)/epoll(7)
accept(2)
read(2)/recv(2)
write(2)/send(2)
close(2)
...
```

## some remarks on Unix I/O

- blocking vs. non-blocking
  - O\_NONBLOCK
  - vgl. synchrone vs. asynchrone API-calls
- I/O multiplexing: select, poll (epoll, kqueue, /dev/poll)
  - frameworks (ACE) + libraries (libevent)

# Networking-I/O control flow



# Properties of Inet-Servers

- high traffic load
- need for “real-time” performance
- extensive I/O
- -> handle blocking calls carefully
- long uptime

# (1) + (2) = threaded Inet-Servers?

- where threads can be helpful:
  - accept -> process: threaded serve-strategies
    - 1.thread-per-request
    - 2.listener/workers
    - 3.leader/followers
  - process: subsequent inet communication
    1. asynchronous operations
    2. wrap synchronous/blocking operations
- things to avoid [3]: context switches, data copies, memory allocation, (lock contention)

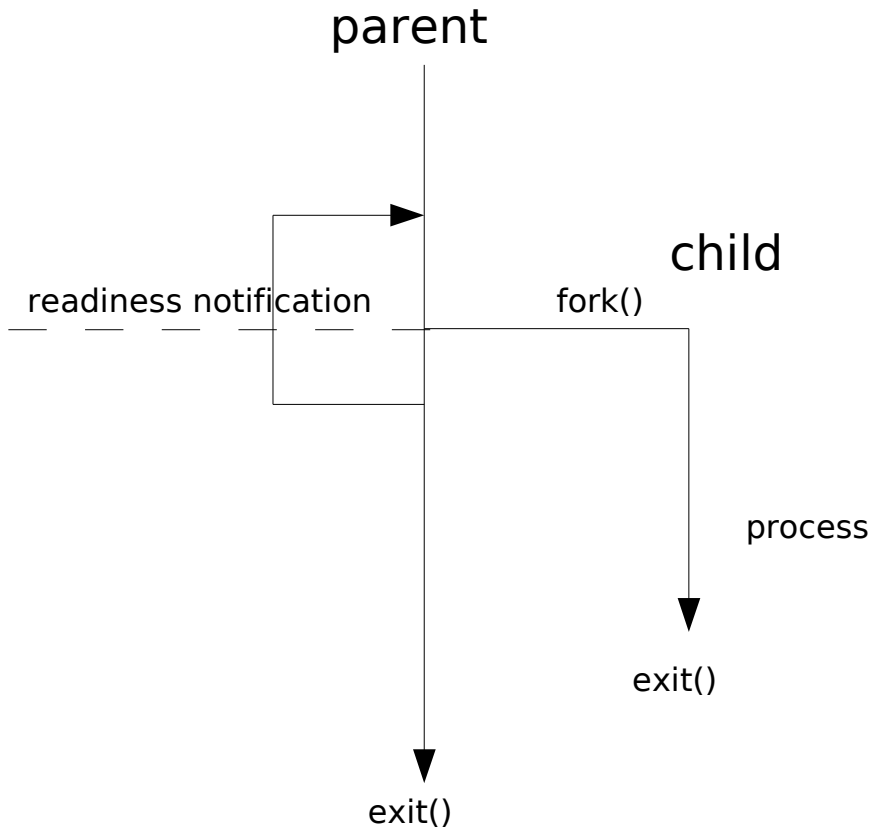
## 2. Threaded Serve-Strategies

# Serve-Paradigmen

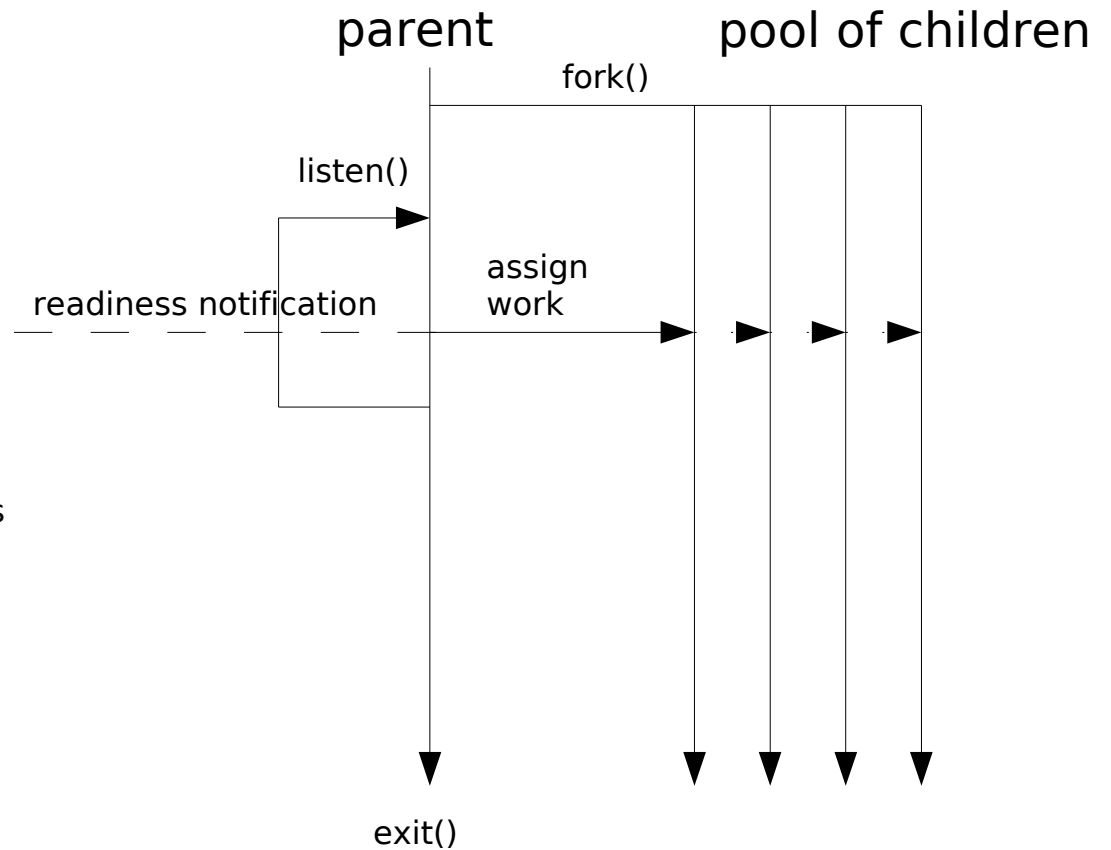
1. iterative (single threaded, single process)
2. forked, preforked
3. threaded (1): thread-per-request
4. threaded (2): listener/workers
5. threaded (3): leader/follower

# 1. + 2. fork()

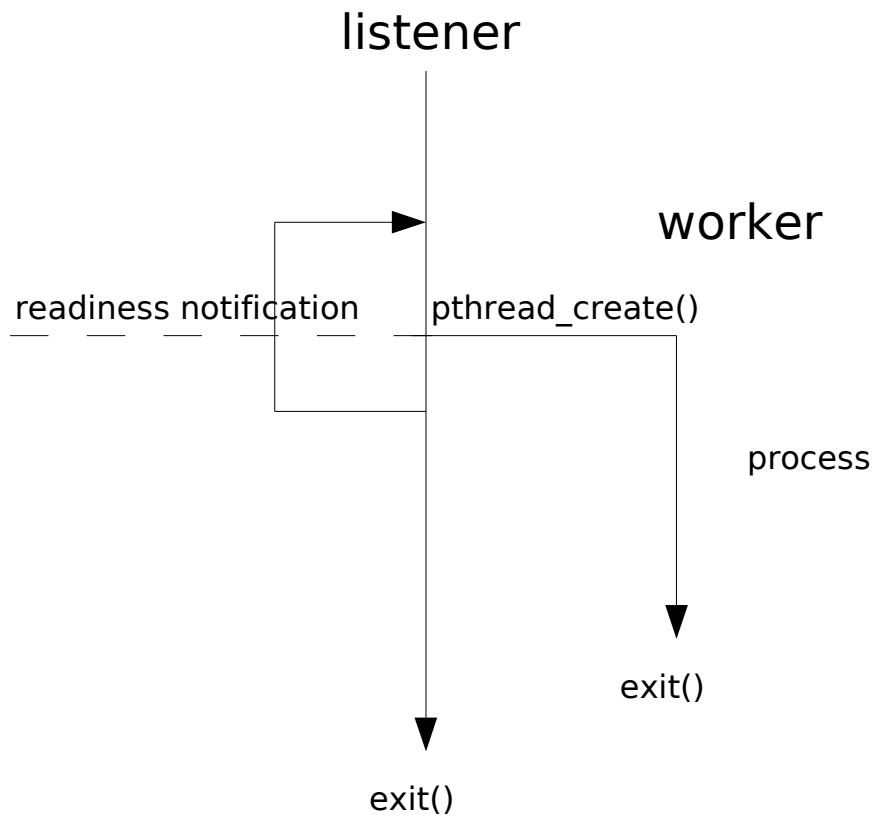
## forked



## preforked

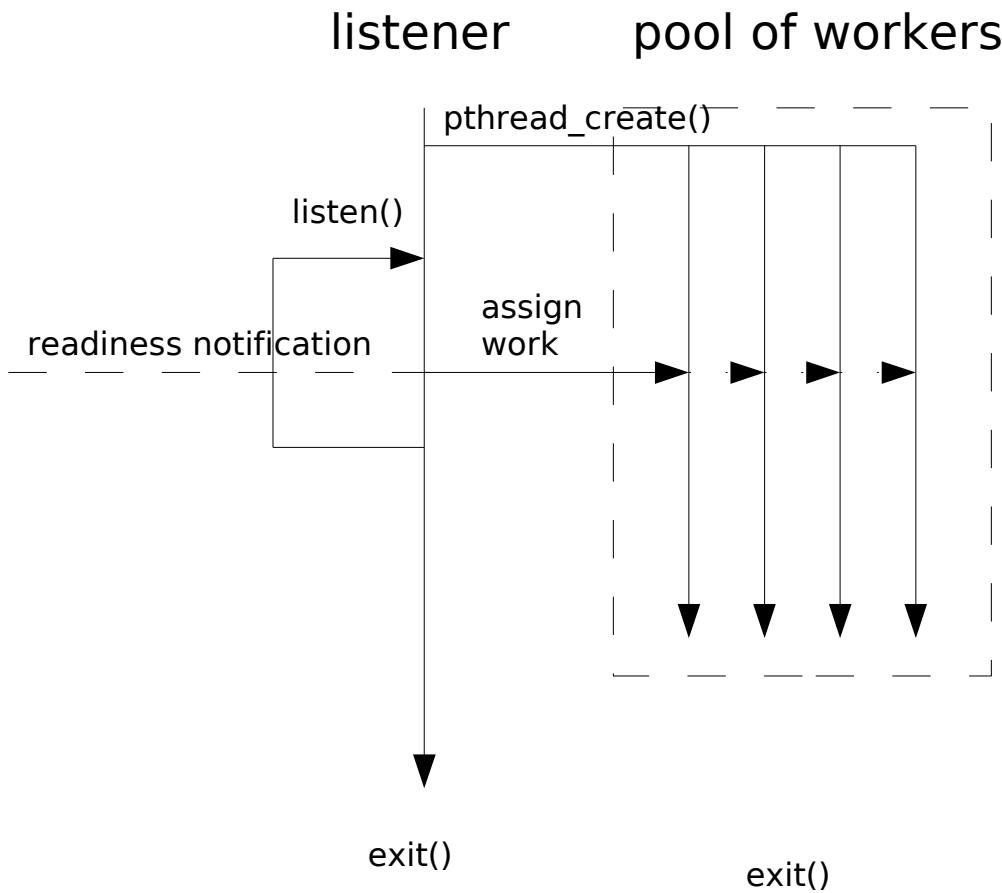


# 3. thread-per-request



- + less resources
- creation overhead
- context switch ...
- + ... but cheap

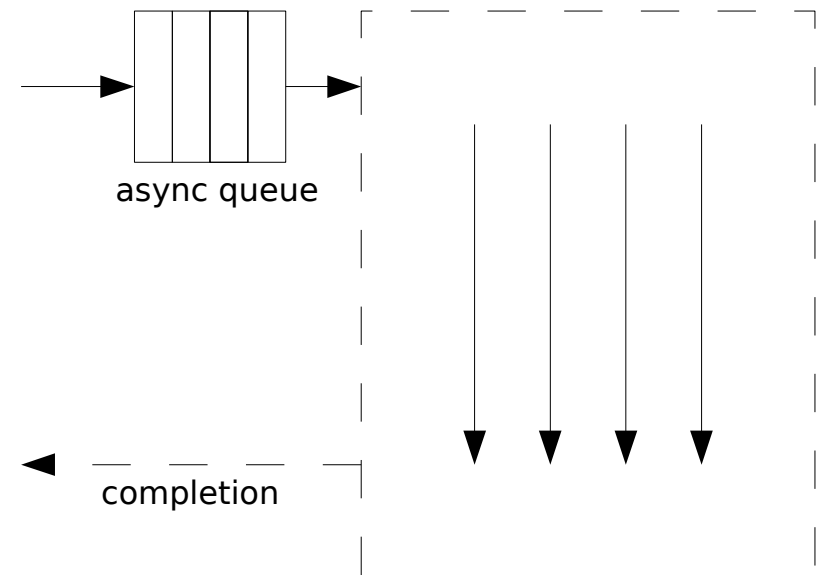
# 4. prethreaded



- + less resources
- + initial creation
- context switch with overhead
  
- + separation of responsibilities (locking)

# Thread Pools (1): listener/workers

- Pool =
  - request-handles
  - asynchrone queue
  - array of threads
- completion
  - do it (z.B. callback)
  - pass back (queue)
  - notify (s.u., asynchron)



# Thread Pools (1)

```
struct async_q {
    size_t nmax;
    size_t ncur;

    struct qnode *head;
    struct qnode *tail;

    pthread_cond_t  notEmpty;
    pthread_mutex_t lock;
};
struct qnode* enqueue(struct async_q *q,
    struct qnode* new_node);
struct qnode* dequeue(struct async_q *q);
```

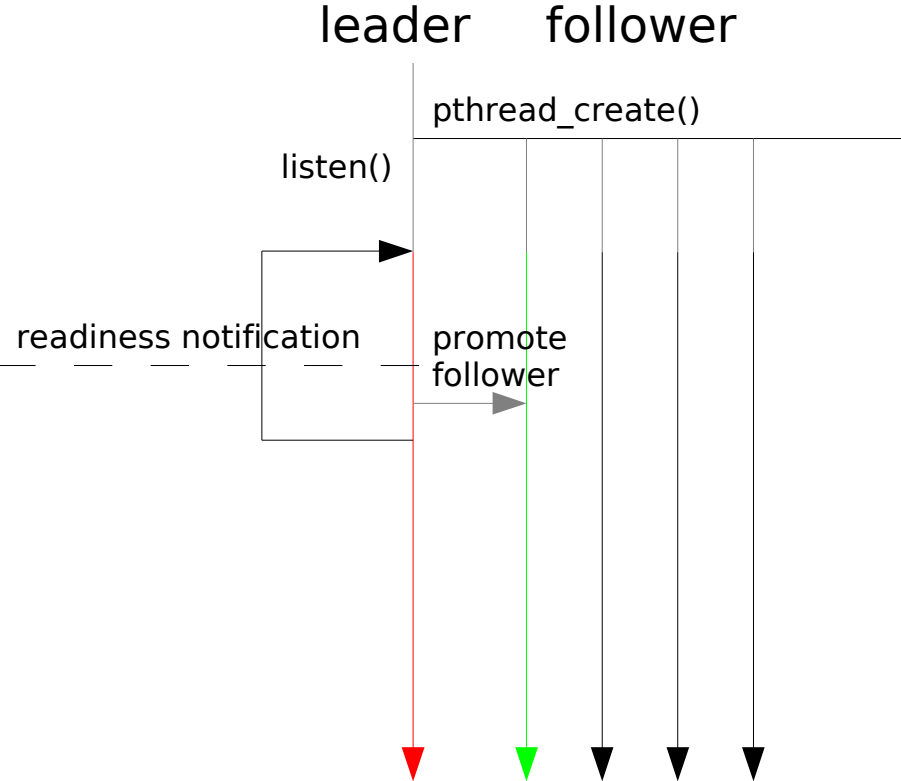
```
struct qnode {
    void *user_data;
    struct qnode *next;
};
```

```
worker_thread:
```

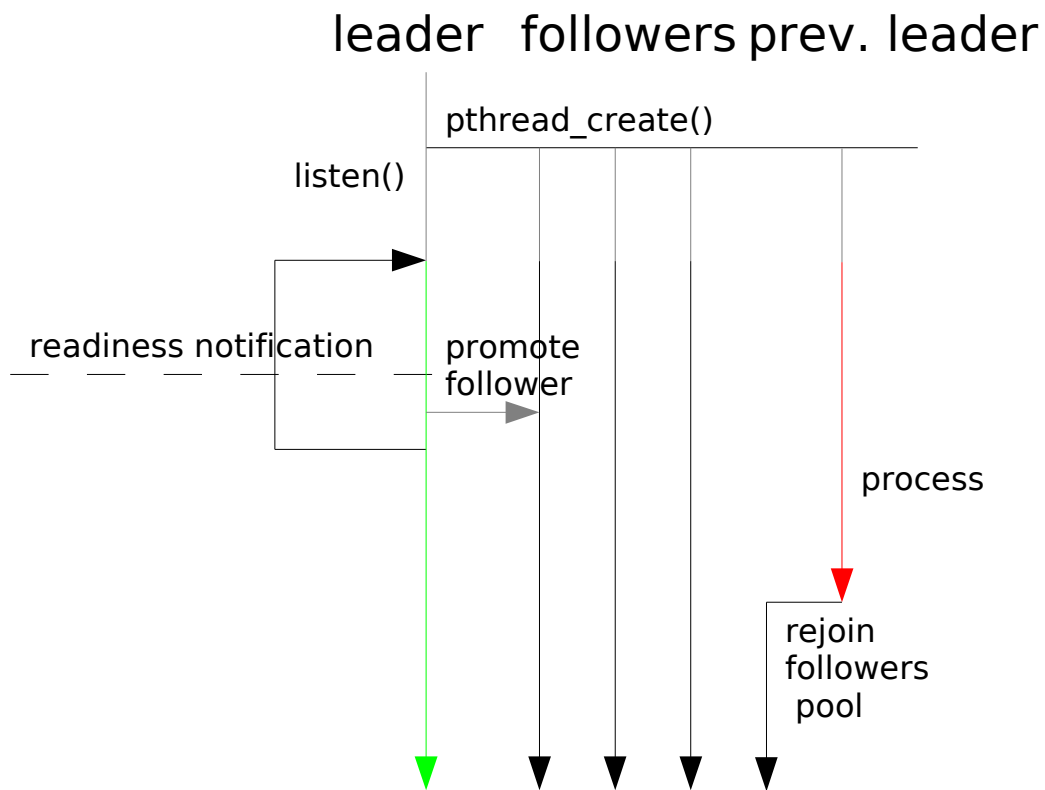
```
while (1) {
    pthread_mutex_lock(&(queue->mut));
    if (queue->cur <= 0)
        pthread_cond_wait(&queue->notEmpty, &queue->lock);

    work = dequeue(queue);
    pthread_mutex_unlock(&(queue->mut));
    process(work);
    //...
}
```

# 5. leader/followers



# leader/followers (cont.)



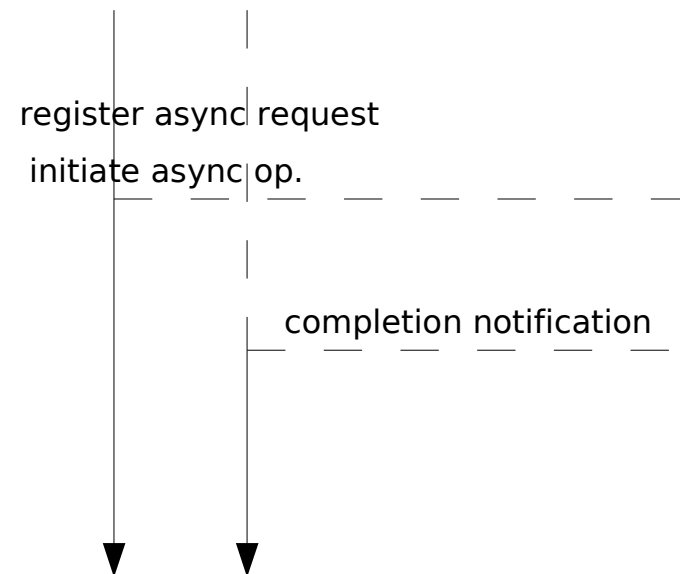
- + less resources
- + initial creation
- + no context switch
- architecture

## 3. Subsequent Network Communication

# async operations

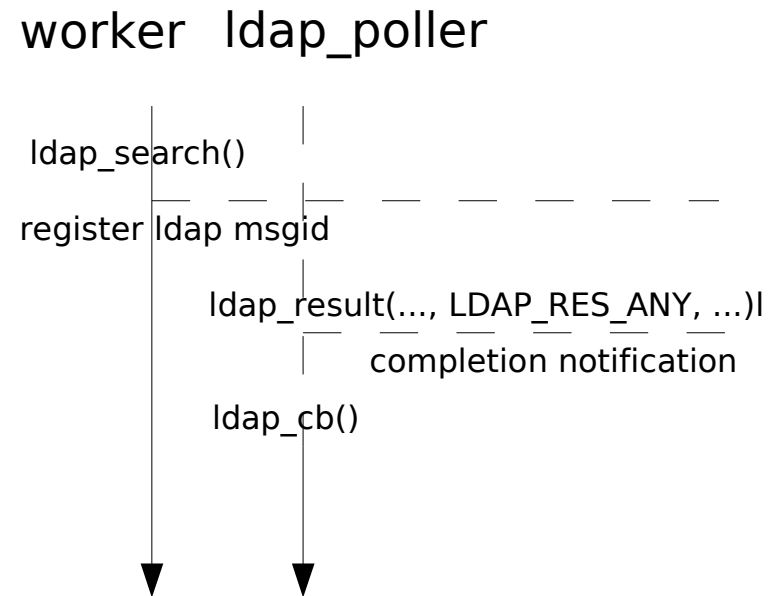
- potentially long-duration non-blocking ops
- e.g. network communication
  - `ldap_search()` -> `ldap_result()`
  - `send()` -> `recv()`
- building blocks:
  - async op handles
  - initiation
  - completion

initiator poller



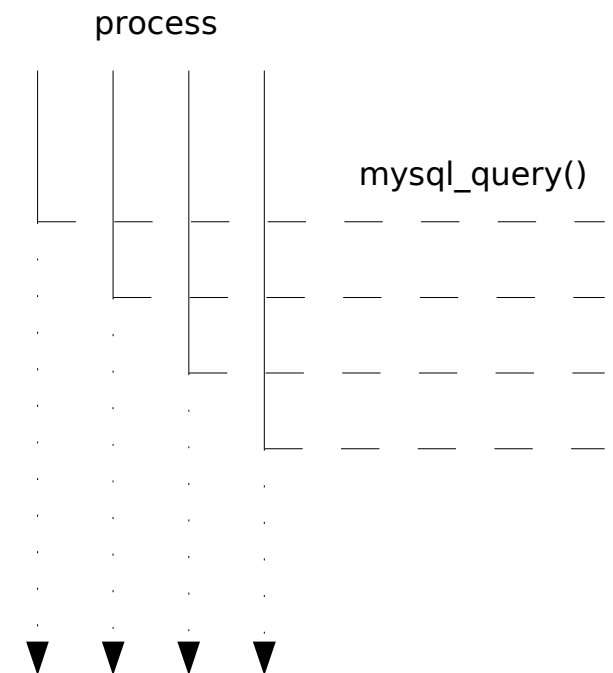
# async ops (cont.)

- async handles:
  - fd
  - completion handler
- completion:
  - poll (active?)
  - demultiplex handle
  - continue processing
- notification mechanisms
  - fd-activity (POLLIN; API z.B. ldap\_result())
  - signal

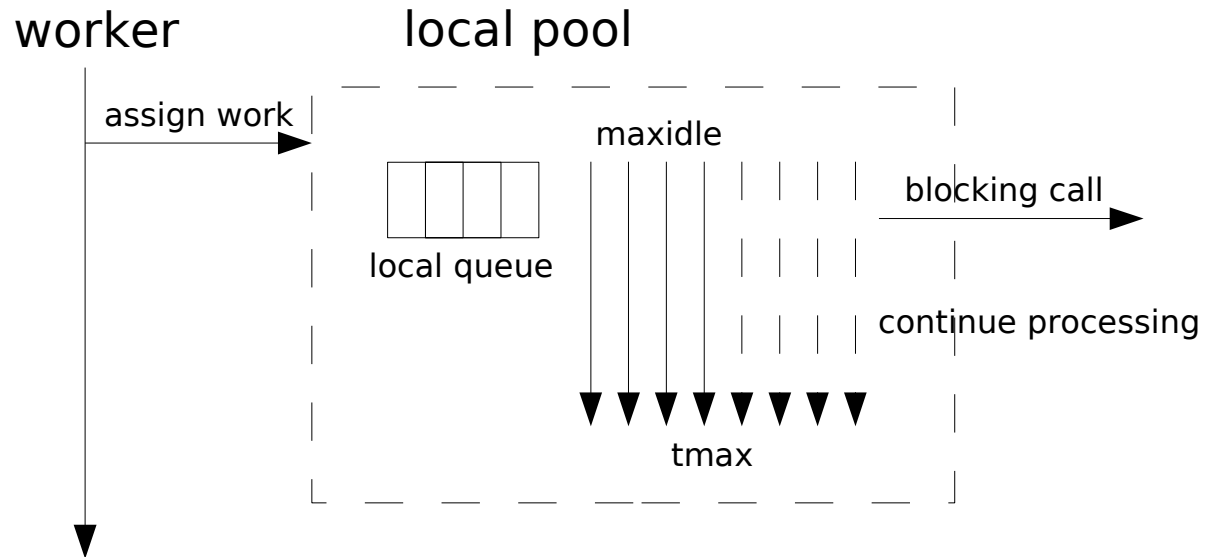


# synchronous operations

- potentially long duration, blocking
- e.g. network communication:
  - `int mysql_query(MYSQL *mysql, const char *query);`
  - `ldap_search_s(...)`
- worst case: unresponsive server
- one solution: thread pool



# Thread Pools (2)



**assign work:**

```
tmax threads reached on local pool?  
yes: try to put request on local-q  
no : idle thread available?  
yes: assign request to idle thread  
no : spawn new thread + assign work
```

## Thread Pools (2)

```
struct qnode {
    void *user_data;
    int (*exec)(void *user_data);
    void (*cb)(void *user_data,
               int ret);
    struct qnode *next;
};
```

```
struct tnode {
    struct tnode *next;
    struct queue *queue;
    struct qnode *work;

    pthread_mutex_t lock;
    pthread_cond_t haveWork;
}
```

credits to aschultz

```
static void *worker_thread(void *arg)
{
    struct tnode *n = (struct tnode *)arg;
    pthread_mutex_lock(&n->lock);

    while (42) {
        /* no work? -> become idle! */
        if (!n->work)
            pthread_cond_wait(&n->haveWork, &n->lock);

        /* process work */
        int ret = n->work->exec(n->work->user_data);
        n->work->cb(n->work->user_data, ret);
        free(n->work);

        /* have work on our queue? */
        pthread_mutex_lock(&n->queue->mutex);
        n->work = dequeue(n->queue);
        pthread_mutex_unlock(&n->queue->mutex);
        if (!n->work) /* no more work
                    -> become idle or terminate */
            if (!dealloc_thread(n))
                break;
    }
    shutdown_thread(n);
    return NULL;
}
```

# References

- [1] W. R. Stevens et al., UNIX® Network Programming Volume 1, Third Edition: The Sockets Networking API, Addison Wesley, 2003
- [2] Buschmann, F., Schmidt, D. C.; Pattern oriented software architecture, Vol. 2: Patterns for Concurrent and Networked Objects; Wiley; 2000
- [3] High-Performance Server Architecture:  
<http://pl.atyp.us/content/tech/servers.html>
- [4] Pthreads specification:  
<http://www.opengroup.org/onlinepubs/000095399/idx/threads.html>
- [5] Dan Kegel, The C10K Problem: <http://www.kegel.com/c10k.html>
- [6] Nichols, B. et al.; Pthreads Programming; O'Reilly; 1998

Thanks