CS 744 DECS Lecture 18
A Performance Analysis View of Concurrency
(Multithreading and Locks)

Autumn 2024 Guest Lecture: Varsha Apte

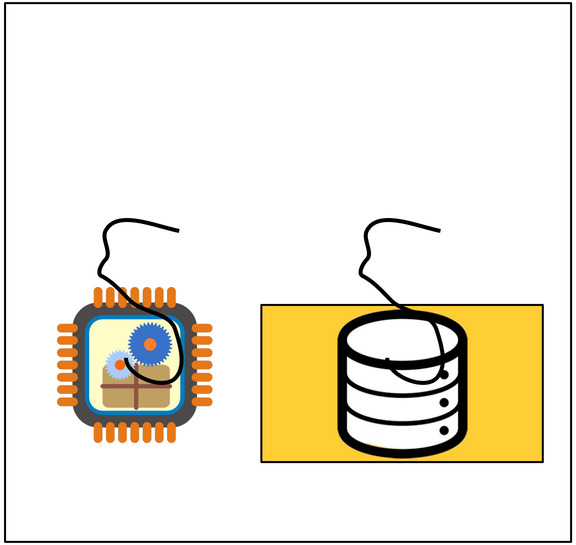
Recap: Multithreading: How? pthread library

```c
#include <stdio.h>
#include <pthread.h>
#include <assert.h>
#include <stdlib.h>

typedef struct __myarg_t {
    int a;
    int b;
} myarg_t;

typedef struct __myret_t {
    int x;
    int y;
} myret_t;


void *mythread(void *arg) {
    myarg_t *m = (myarg_t *) arg;
    printf("%d %d\n", m->a, m->b);
    myret_t *r =
malloc(sizeof(myret_t));
    r->x = 1;
    r->y = 2;
    return (void *) r;
}

```

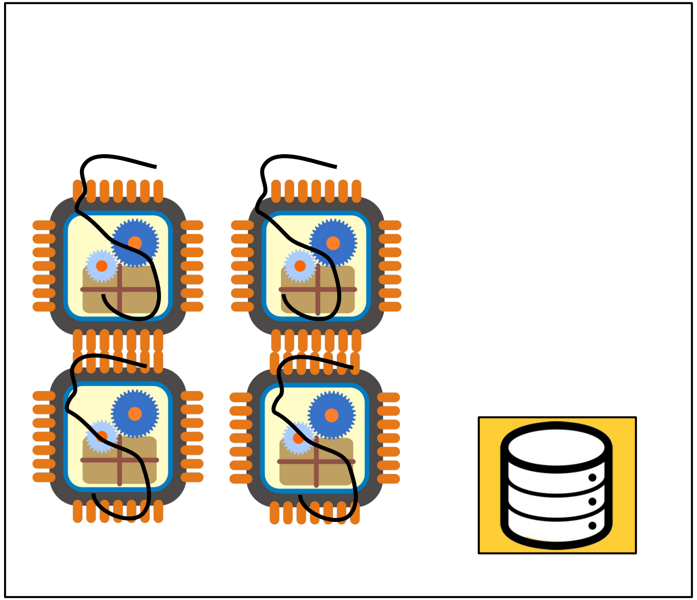
Recap: Multithreading: How? pthread library (Cont.)

```
int main(int argc, char *argv[]) {
    int rc;
    pthread_t p;
    myret_t *m;

    myarg_t args;
    args.a = 10;
    args.b = 20;
    pthread_create(&p, NULL, mythread, &args);
    pthread_join(p, (void **) &m);  // this thread has been
                                     // waiting inside of the
// pthread_join() routine.
    printf("returned %d %d\n", m->x, m->y);
    return 0;
 }
```

Multithreading - why?

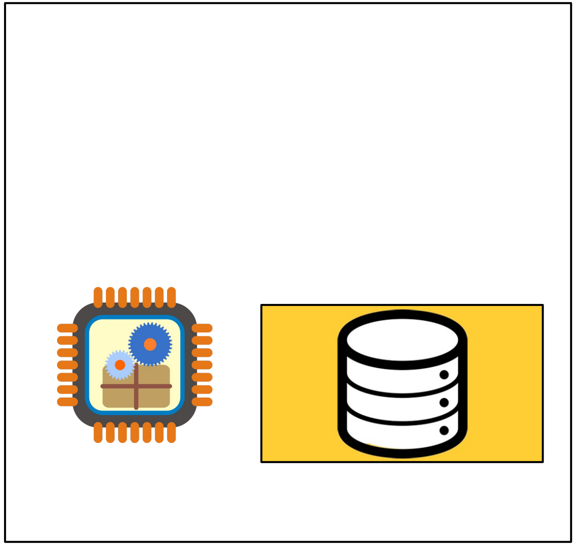


Why when Single-core -
One thread can use the CPU
while other waits for I/O

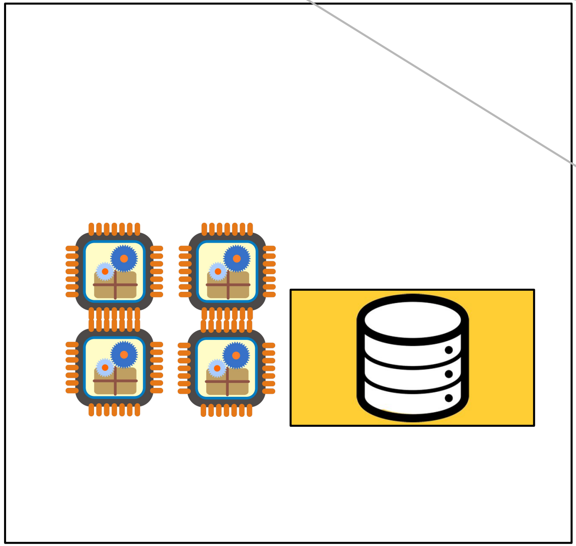


Why when multi-core -
Multiple threads can run
parallely

Multithreading - why?



Why when
Single-core



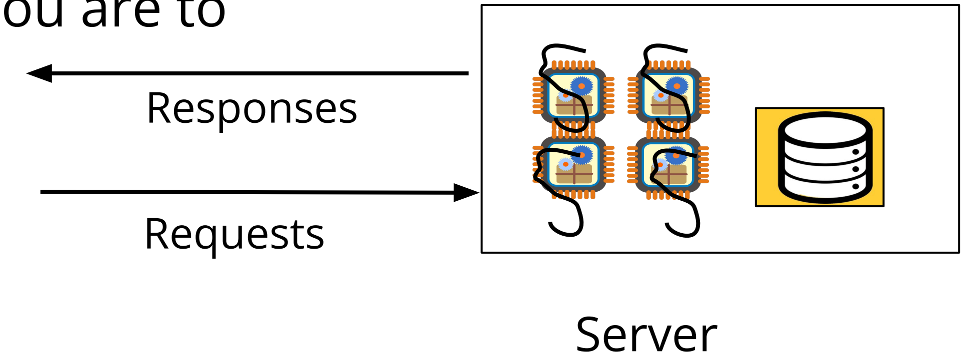
Why when
multi-core

To decrease the time taken for work or increase the 'rate' at which work is done
By making better use of resources

Suppose you are asked to build a server

Server gets some request from a client, does some work, returns a response. You are to build a server that will

- Give 'good performance'
- 'scale'



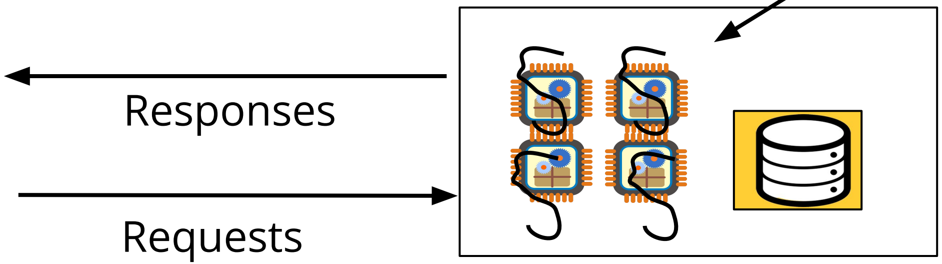
What do these words mean?

How will you use multithreading for "good performance and scalability"?

How will you *know* your server is giving "good performance and scalability"?

First
Metrics for describing server Performance

Throughput (responses per second)

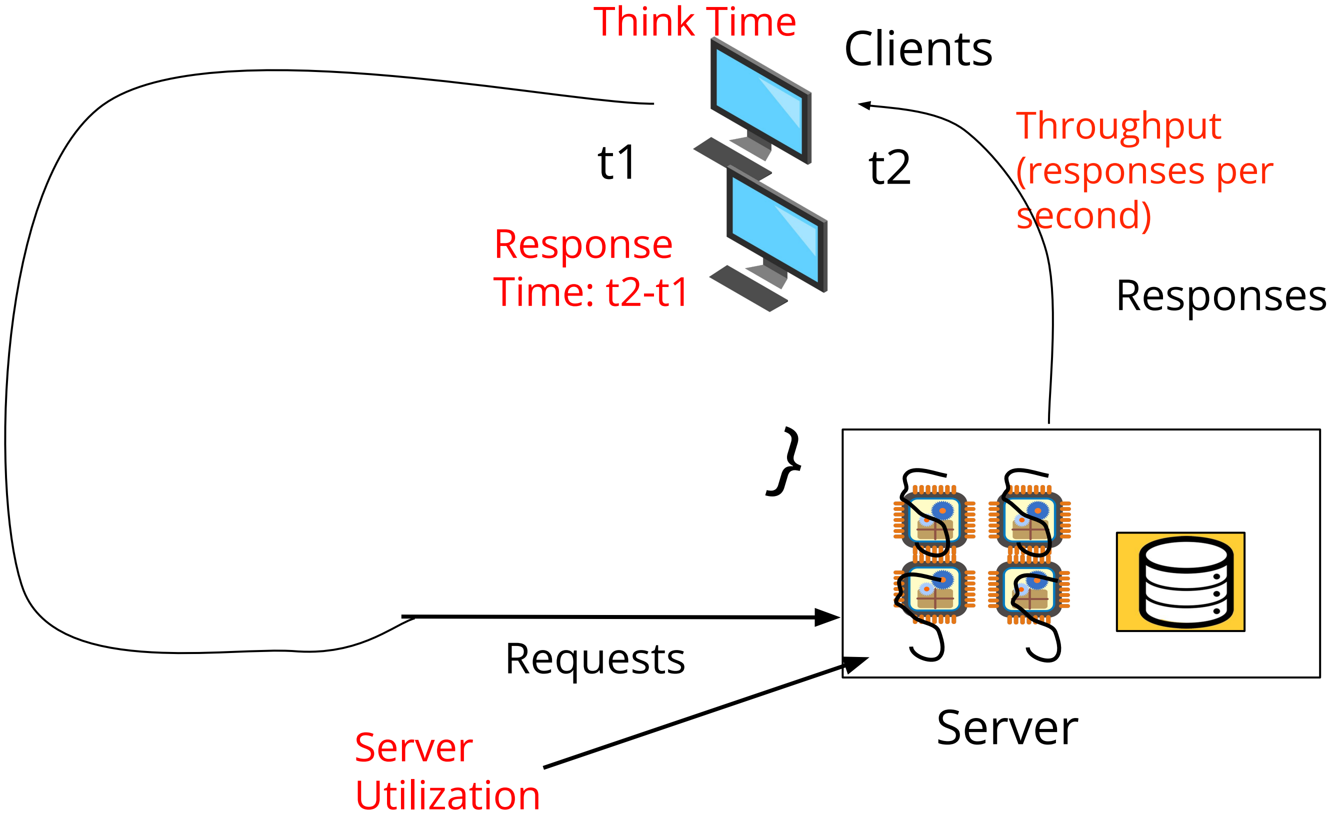


Server

Server Utilization - average fraction of time the server resource is busy (eg, server CPU utilization)

Performance delivered is a function of server properties and "*load*" *parameters*

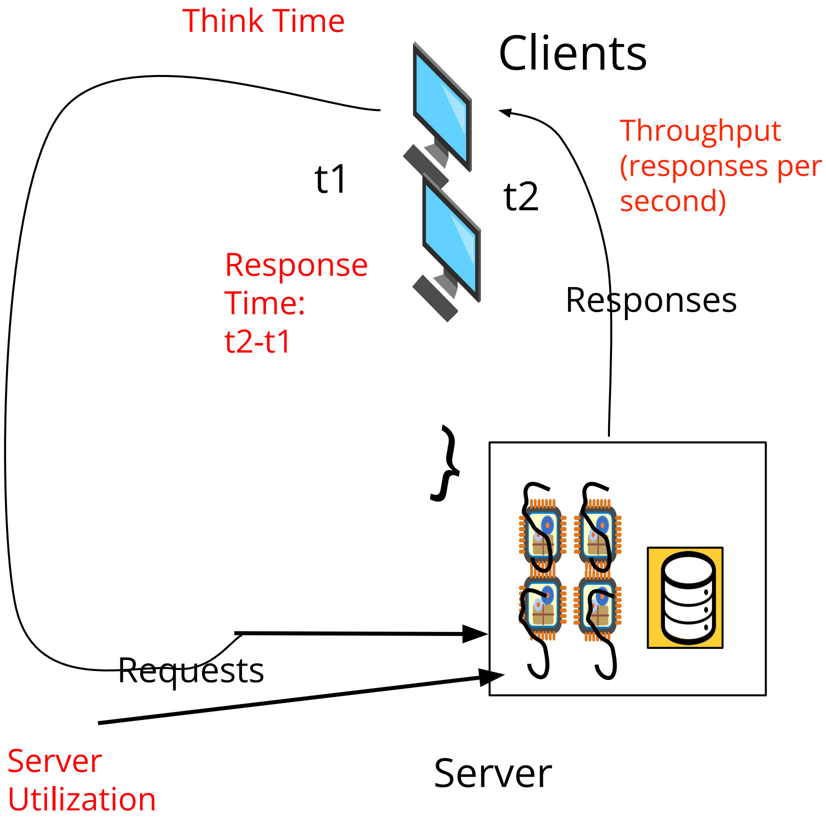
Load: Closed Loop view of a Client-server system



Clients are in a *closed request-response loop with the server - Each* Client issues request, waits for response 'thinks' then issues next request

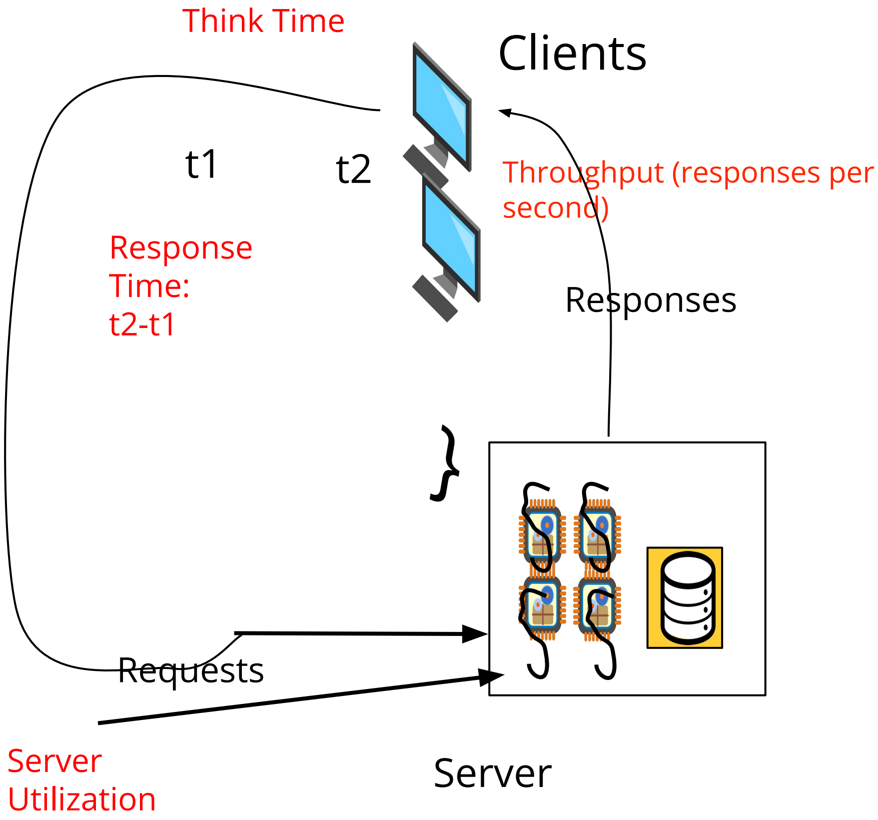
Load parameters: Number of clients, think time, the type of requests

Performance questions that can be asked



- If there are M clients whose average think time is **γ**,
  - what is the total server throughput?
  - What is the response time experienced by clients?
  - What is the server utilization?
- What is the maximum number M* of clients that this server can ‘support’?
- What is the maximum throughput capacity of the server?

Scalability questions that can be asked



- How does the throughput capacity of the server improve if we give more resources ?
- E.g. *Multicore scalability*
  - If I double the number of cores in the server, will throughput capacity double?

Back to… Multithreading design options

- Listener thread that listens for new requests
- Accepts connection, starts `worker' thread, gives the connection to the worker thread
- Worker thread does the work, sends the response
- Options:
  - Thread per request - does entire request (easier, more common)
  - Thread per 'stage of work' (needs some state management across stages)

Number of worker threads

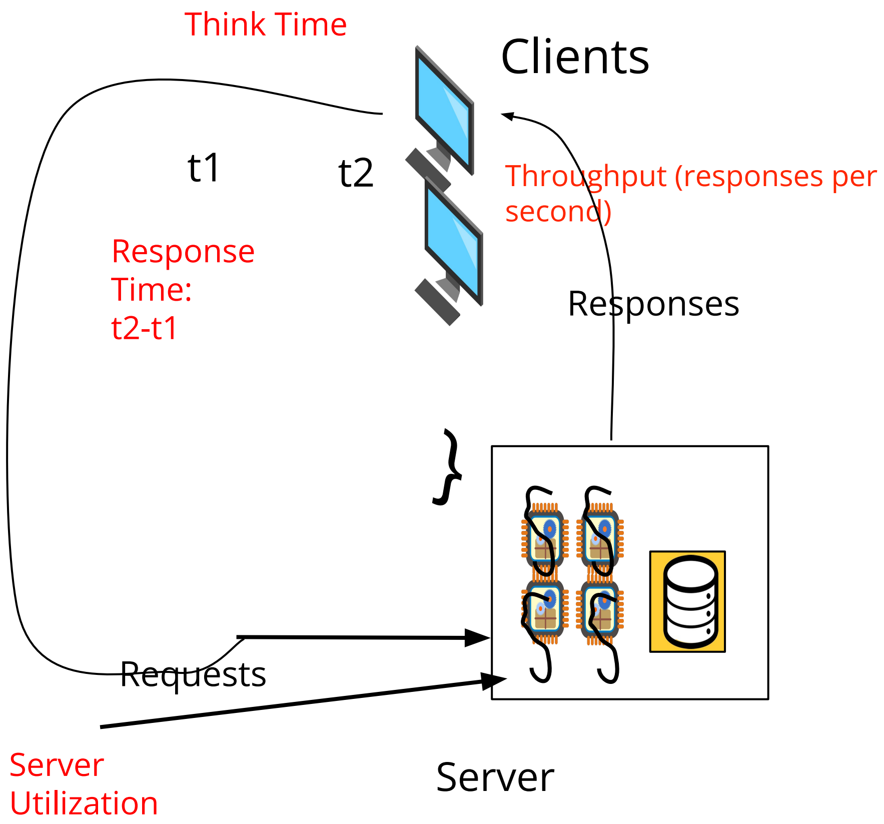
- Create-destroy approach: Create a worker thread for each request, thread exits after response is sent
  - Easier
- Thread pool model:
  - Maintain a ‘pool’ of threads.
  - Maintain a shared queue of requests
  - Thread remains after response is sent, picks up the next available request in the queue, or waits (idle) for the next request

How can we decide which design is good?

How many threads to configure?

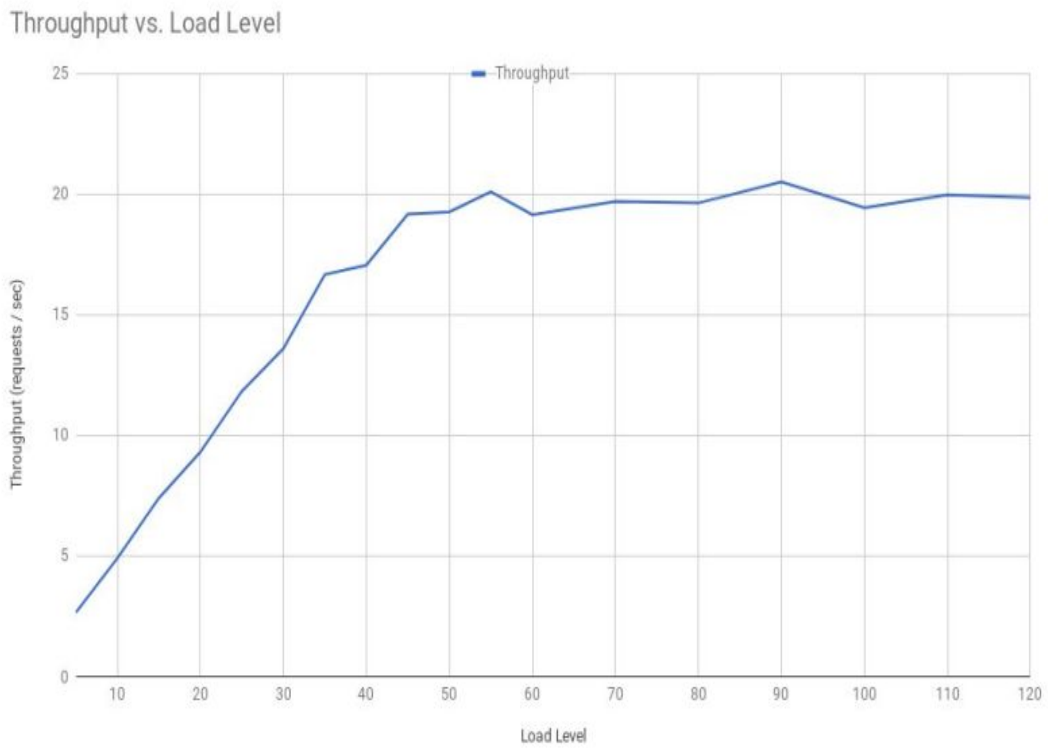
And so on...

One way: Run 'Load Tests' and measure performance



- Set up a client-server test bed
- Emulate multiple clients
  - M clients
  - Think time γ
- Measure average throughput, utilization, response time as a function of increasing load - typically number of clients M = 1 to some max

Example throughput vs Number of Users

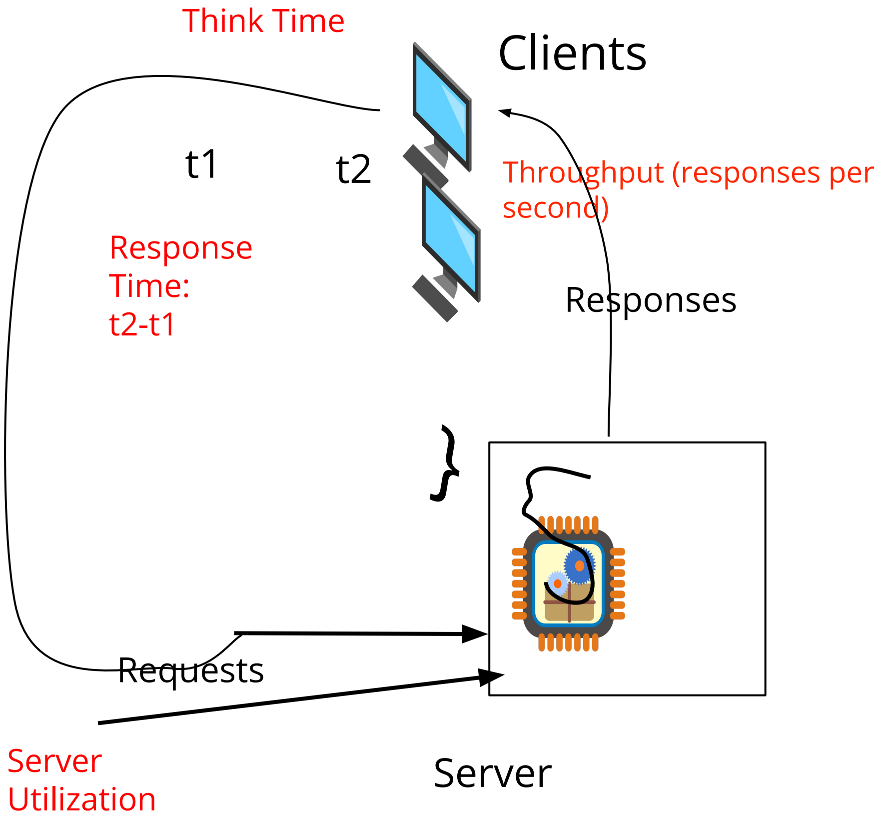


How do we know the experiment is correct?

How do we know if this is 'good' or 'bad' performance?

Can we estimate other metrics in other scenarios based on this much ?

Basic reasoning for Performance in Closed Loop Systems



Consider CPU-bound requests, single core, single thred

Processing time per request is known = τ

We can estimate asymptotes of the performance graphs

Performance metrics for closed loop experiments

Example: τ = 100 ms, single thread, single core. Think time of clients = 1 sec (1000 ms)

Maximum throughput capacity of the server? How many clients can be supported?
Throughput at M = 1? Throughput at M = 5? At M = 20?
Server Utilization at M = 1? At M = 5? At M = 20?
Response Time at M = 1? At M = 20?

Performance metrics for closed loop experiments

In general, if service time (of bottleneck resource) is τ,

max throughput for one resource = $1/\tau$ requests/sec

Let Response Time when M clients = R(M)

Throughput $\Lambda = M/(R(M) + \gamma)$

$R(M) = M/\Lambda - \gamma$

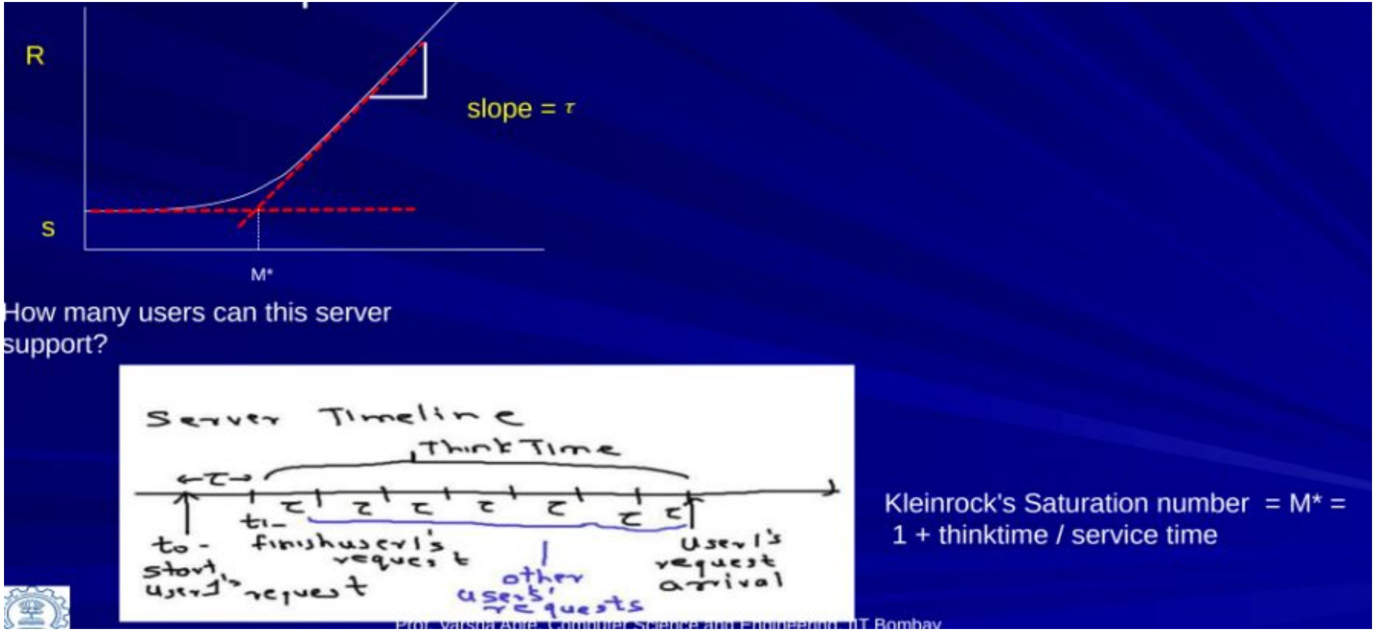
As M increases Throughput $\Lambda \rightarrow 1/\tau$ requests/sec

$R(M) \rightarrow M\tau - \gamma$ (Slope is τ)

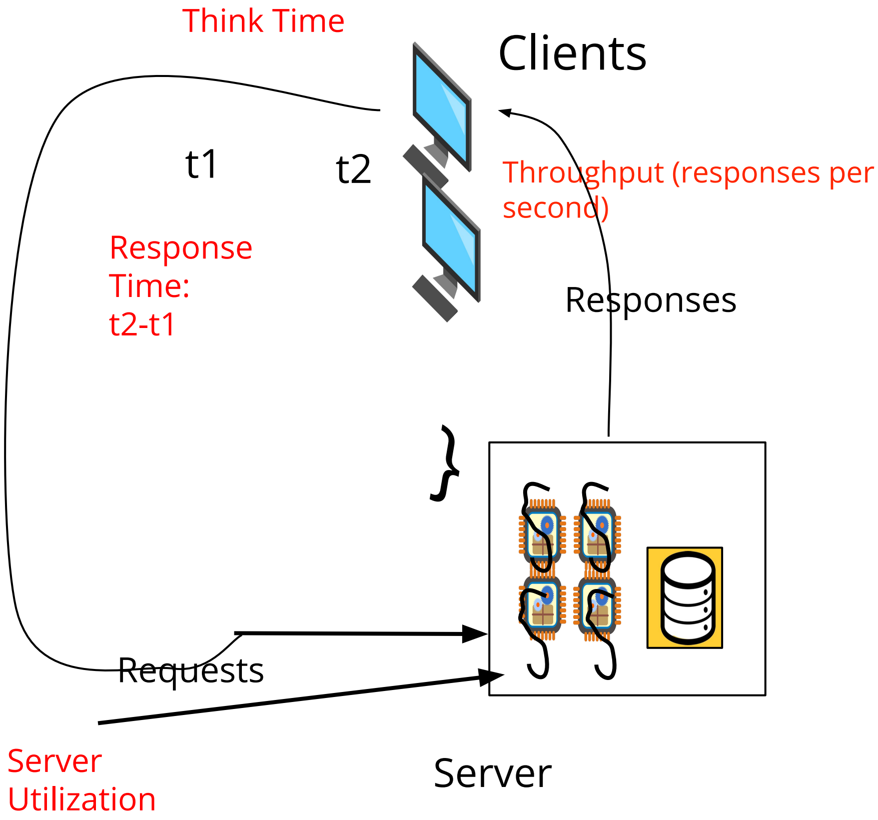
Server utilization $\rho = \Lambda \times \tau \rightarrow 1$ as M increases

Number of clients that can be supported

Heuristic: 1 + Thinktime/servicetime per core (or thread)



Basic Reasoning: Multithreaded, multi core setup



Consider CPU-bound requests, multi core, multi thread, NO LOCKS required

Processing time per request is known = τ

We can estimate asymptotes of the performance graphs

Performance metrics for closed loop experiments (multithread, multicore)

Example: **τ** = 100 ms, **four threads, four cores**. Think time of clients = 1 sec (1000 ms)
Maximum throughput capacity of the server? How many clients can be supported?
Throughput at M = 1? Throughput at M = 5? At M = 100?
Server Utilization at M = 1? At M = 5? At M = 100?
Response Time at M = 1? At M = 100?

Performance metrics for closed loop experiments

In general, if service time is τ, number of cores = c

max throughput= c/τ requests/sec

Let Response Time when M clients = R(M)

Throughput $\Lambda = M/(R(M) + \gamma)$

$R(M) = M/\Lambda - \gamma$

As M increases Throughput $\Lambda \rightarrow c/\tau$ requests/sec

$R(M) \rightarrow M\tau/c - \gamma$ (Slope is τ/c)

Server utilization $\rho = \Lambda \times \tau / c \rightarrow 1$ as M increases

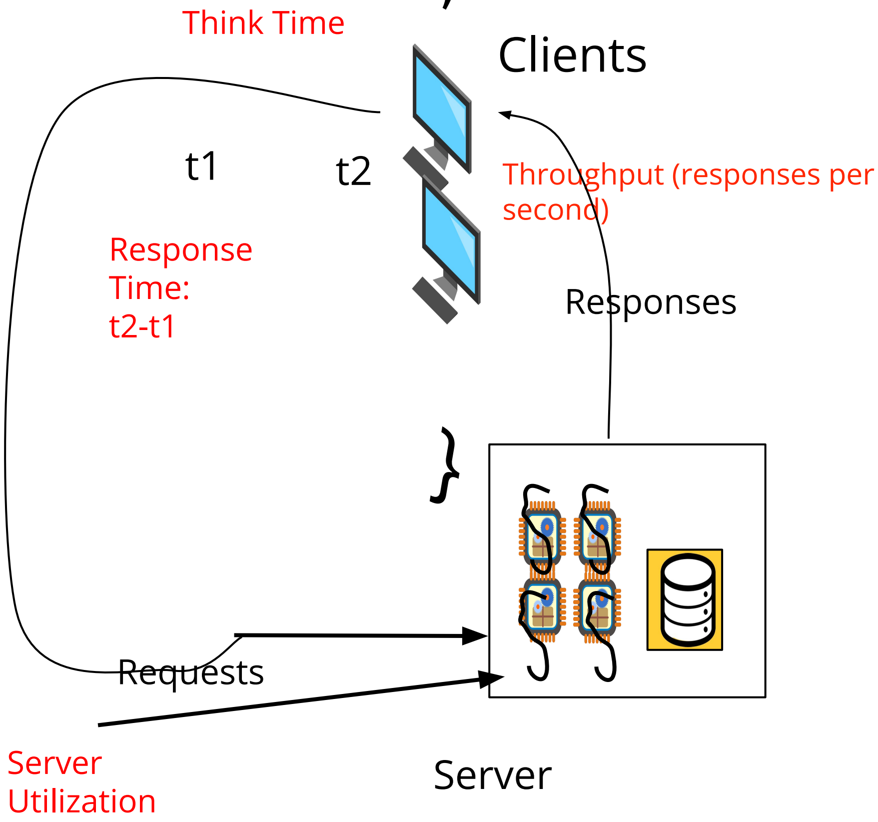
Max Number of clients: $\gamma c / \tau$

Case Study - 1

Experimental Performance Measurement of a Web Server (closed load)

Basic Reasoning:

Multithreaded, multi core setup, sync bottleneck



Processing time per request is known = **τ = τ1 + τ2** where **τ2** ms are executed under a mutex lock.

Performance metrics for closed loop experiments (multithread, multicore, some code under mutex)

Example: **τ** = 100 ms, **eight threads, eight cores**. Think time of clients = 1 sec (1000 ms). **τ**2 = 20 ms

Maximum throughput capacity of the server? How many clients can be supported?
Maximum Server Utilization? Response time behavior?

Max throughput with any number of threads or cores : 1000/20 = 50 reqs/sec
At 50 requests/sec, server utilization = 50 x 100 / (8 * 1000) = 0.625
Slope of Response time asymptote will be 20 ms

Case Study - 2

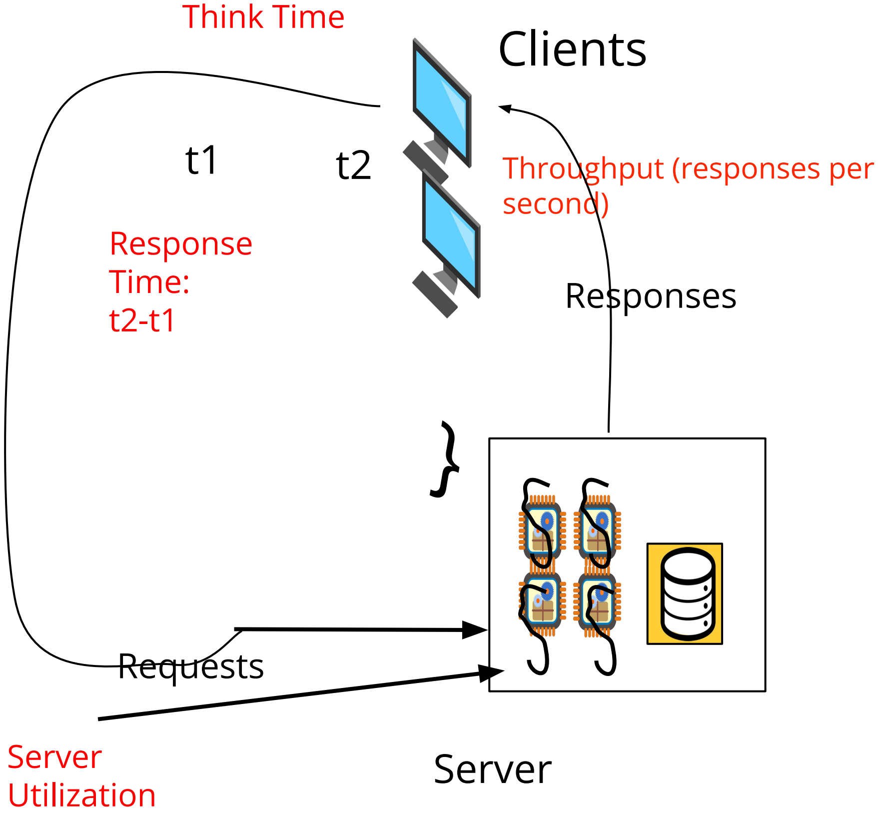
Study of four versions of a “simple” autograding server

https://www.dropbox.com/scl/fi/jn929eapbmib7v975u4im/DECServer_23D0361.pdf?rlkey=3ia4omysij0r0ozvz9d5dvk76&dl=0

Autograding server

- Functionality:
  - Accepts a C++ program for grading
  - Compiles, executes (if compiled successfully), checks output (if executed successfully)
  - Sends back a pass/fail response to client
- Design:
  - V1: Single thread, single core
  - V2: Multithread, create-destroy, multi-core
  - V3: Multithread, thread-pool, multi-core
  - V4: ‘asynchronous design’

Moral of the story?



If results don't actually match 'theoretical expectations", then why bother with estimates/predictions?

Predicting a 'baseline' expected metric helps

- Identifying errors in experiments
- Isolate actual vs assumed bottlenecks
- Idenity myriad other issues in the server

Case Study - 3

Multicore Scalability Bottleneck Analysis of a real Autograding server

https://docs.google.com/presentation/d/1aHdJB7VsxlBVoCfcQM43YJYuCSC8VKzix_eMt4V7PYo/edit?usp=sharing

Thank you

Laws to cover

Amdahl’s law

asymptotes

Throughput law

Utilization law

Little’s law

Response time graph

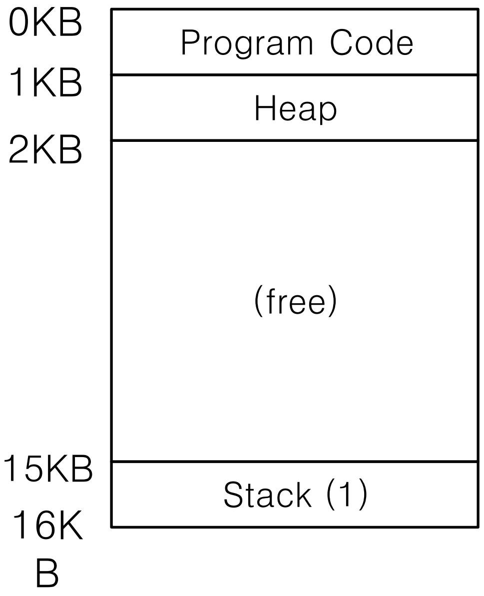
Saturation number

Multithreading - RECAP

Two goals for multithreading

1. Make use of the CPU when a thread blocks on I/O
2. Make use of multiple cores

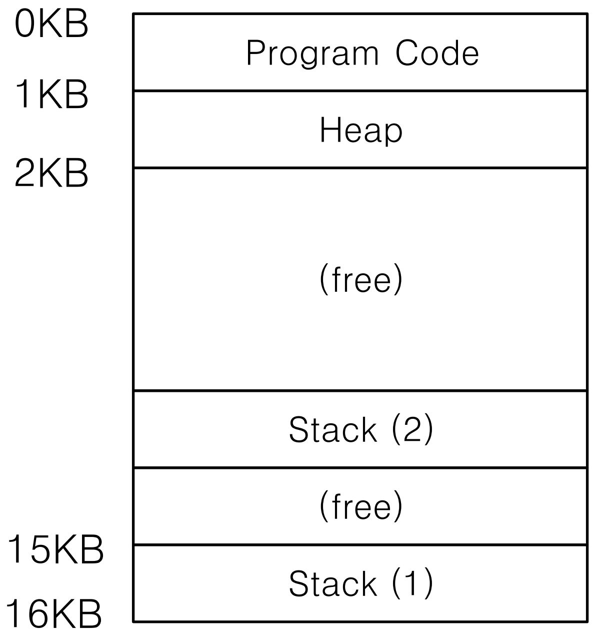
Thread vs Process



The code segment : where instructions live
The heap segment : contains malloc'd data dynamic data structures (it grows downward)

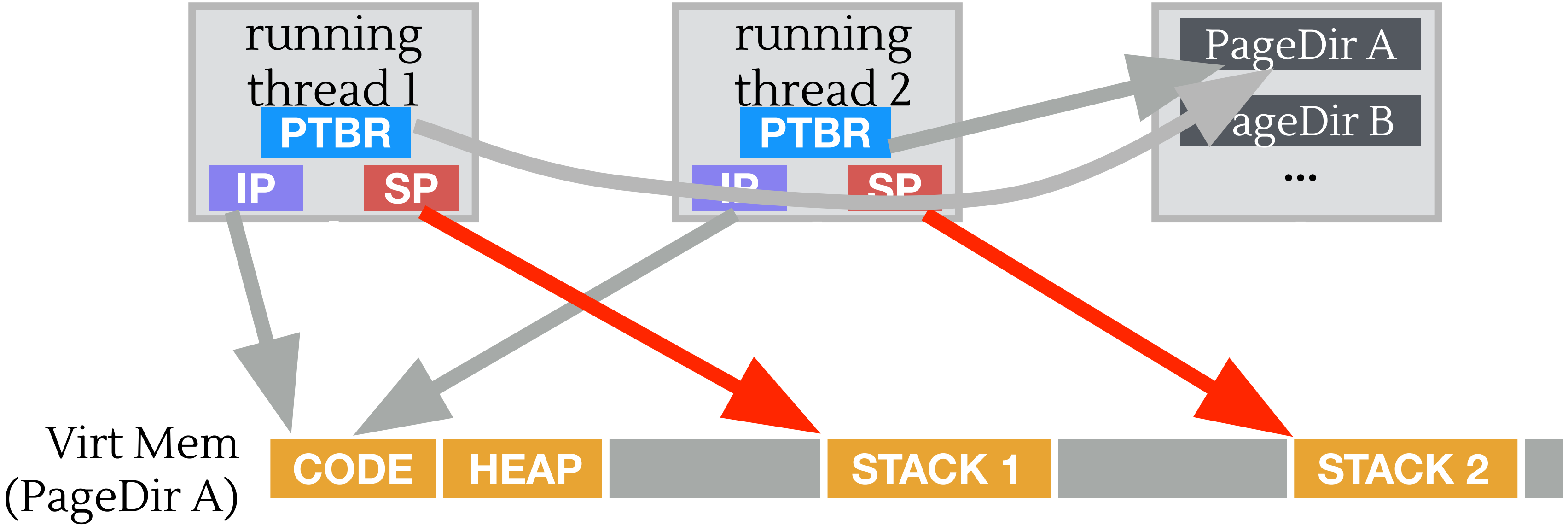
(it grows upward)
The stack segment : contains local variables arguments to routines, return values, etc.

A Single-Threaded Address Space



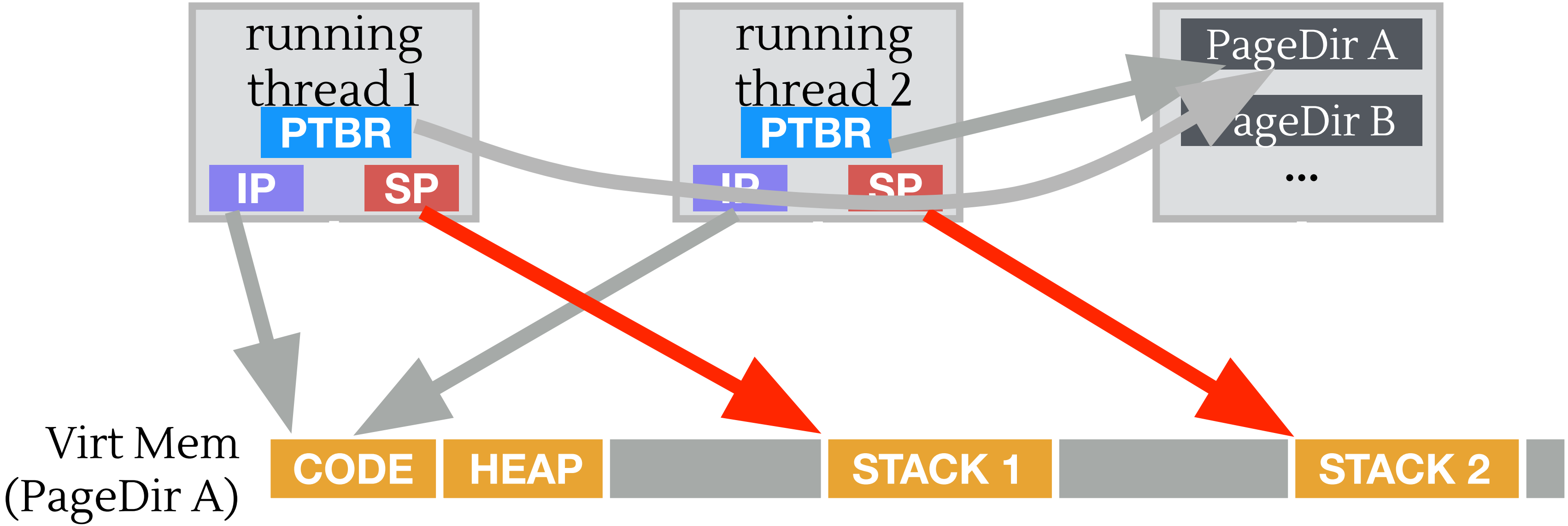
Two threaded Address Space

- There will be one stack per thread.



Share code, but each thread may be executing different code at the same time ⇒
□ Different Instruction Pointers

threads executing different functions need different stacks ⇒
Different stack pointers



⇒ Each thread has its own program counter and set of registers.
One thread control blocks(TCBs) per thread store the state

When switching from running one (T1) to running the other (T2),
The register state of T1 be saved.
The register state of T2 restored.
The address space remains the same.

THREAD VS. Process

Multiple threads within a single process share:

- Process ID (PID)
- Address space
  - Code (instructions)
  - Most data (heap)
- Open file descriptors
- Current working directory
- User and group id

Each thread has its own

- Thread ID (TID)
- Set of registers, including Program counter and Stack pointer
- Stack for local variables and return addresses
(in same address space)

Threads API

POSIX thread library

pthread_create

pthread_join

Thread Creation

- How to create and control threads?

```
#include <pthread.h>

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

- `thread`: Used to interact with this thread.
- `attr`: Used to specify any attributes this thread might have.
  - Stack size, Scheduling priority, …
- `start_routine`: the function this thread start running in.
- `arg`: the argument to be passed to the function (start routine)
  - *a void pointer* allows us to pass in *any type of* argument.

Return value:
0 if creation is successful,
Enum if fail

Example: Creating a Thread

```
#include <pthread.h>

typedef struct __myarg_t {
    int a;
    int b;
} myarg_t;

void *mythread(void *arg) {
    myarg_t *m = (myarg_t *) arg;
    printf("%d %d\n", m->a,
m->b);
    return NULL;
}
```

```
int main(int argc, char *argv[]) {
    pthread_t p;
    int rc;

    myarg_t args;
    args.a = 10;
    args.b = 20;
    rc = pthread_create(&p, NULL,
mythread, &args);
    …
}
```

Wait for a thread to complete

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

- `thread`: Specify which thread *to wait for*
- `value_ptr`: A pointer to the return value
  - Because `pthread_join()` routine changes the value, you need to pass in a pointer to that value.

Example: Waiting for Thread Completion

```c
#include <stdio.h>
#include <pthread.h>
#include <assert.h>
#include <stdlib.h>

typedef struct __myarg_t {
    int a;
    int b;
} myarg_t;

typedef struct __myret_t {
    int x;
    int y;
} myret_t;


```

```c
void *mythread(void *arg) {
    myarg_t *m = (myarg_t *) arg;
    printf(“%d %d\n”, m->a, m->b);
    myret_t *r = malloc(sizeof(myret_t));
    r->x = 1;
    r->y = 2;
    return (void *) r;
}

```

Example: Waiting for Thread Completion (Cont.)

```
int main(int argc, char *argv[]) {
    int rc;
    pthread_t p;
    myret_t *m;

    myarg_t args;
    args.a = 10;
    args.b = 20;
    pthread_create(&p, NULL, mythread, &args);
    pthread_join(p, (void **) &m);  // this thread has been
                  // waiting inside of the
// pthread_join() routine.
    printf("returned %d %d\n", m->x, m->y);
    return 0;
}
```

Example: Dangerous code

- Be careful with <u>how values are returned</u> from a thread.

```
void *mythread(void *arg) {
    myarg_t *m = (myarg_t *) arg;
    printf("%d %d\n", m->a, m->b);
    myret_t r; // ALLOCATED ON STACK: BAD!
    r.x = 1;
    r.y = 2;
    return (void *) &r;
}
```

- When the variable `r` returns, it is automatically de-allocated.

Example: Simpler Argument Passing to a Thread

- Just passing in a single value

```
void *mythread(void *arg) {
    int m = (int) arg;
    printf("%d\n", m);
    return (void *) (arg + 1);
}

int main(int argc, char *argv[]) {
    pthread_t p;
    int rc, m;
    pthread_create(&p, NULL, mythread, (void *) 100);
    pthread_join(p, (void **) &m);
    printf("returned %d\n", m);
    return 0;
}
```

LOCKS and CONDITION VARIABLES

Multithreaded programming with shared data

pthread_mutex_lock

pthread_mutex_unlock

pthread _cond

Example

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include "common.h"
#include "common_threads.h"

int max;
volatile int counter = 0; // share

void *mythread(void *arg) {
    char *letter = arg;
    int i; // stack (private per thre
    printf("%s: begin [addr of i: %
&i);
    for (i = 0; i < max; i++) {
        counter = counter + 1; // shared:
    }

    printf("%s: done\n", letter);
    return NULL;
}
```

```c
int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "usage: main-first <loopcount>\n");
        exit(1);
    }
    max = atoi(argv[1]);

ter = %d] [%x]\n",

nter);
L, mythread, "A");
L, mythread, "B");
    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: done\n [counter: %d]\n [should: %d]\n",
        counter, max*2);
    return 0;
}
```

Critical Section. Need to ensure *no interleaving in the read-increment-store sequence of commands.* Only *one* thread should execute all *atomically*.

Locks

- Provide mutual exclusion to a critical section
  - Interface

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

  - Usage (w/o *lock initialization* and *error check*)

```
pthread_mutex_t lock;
pthread_mutex_lock(&lock);
x = x + 1; // or whatever your critical section is
pthread_mutex_unlock(&lock);
```

  - No other thread holds the lock: the thread will acquire the lock and enter the critical section.
  - If another thread hold the lock: the thread will not return from the call until it has acquired the lock.

Locks (Cont.)

- All locks must be properly initialized.
  - One way: using `PTHREAD_MUTEX_INITIALIZER`

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

  - The dynamic way: using `pthread_mutex_init()`

```
int rc = pthread_mutex_init(&lock, NULL);
assert(rc == 0); // always check success!

NULL: mutex attributes field (advanced, skipping)
```

Locks (Cont.)

- Check errors code when calling lock and unlock
  - An example wrapper

```
// Use this to keep your code clean but check for failures
// Only use if exiting program is OK upon failure
void Pthread_mutex_lock(pthread_mutex_t *mutex) {
    int rc = pthread_mutex_lock(mutex);
    assert(rc == 0);
}
```

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_timelock(pthread_mutex_t *mutex,
                           struct timespec *abs_timeout);
```

- These two calls are used in lock acquisition

Locks (Cont.)

- These two calls are also used in lock acquisition

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_timelock(pthread_mutex_t *mutex,
                           struct timespec *abs_timeout);
```

- `trylock`: return failure if the lock is already held
- `timelock`: return after a timeout or after acquiring the lock

Condition Variables

- **Condition variables** are useful when some kind of signaling must take place between threads.

```
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
```

- `pthread_cond_wait`:
  - Put the calling thread to sleep.
  - Wait for some other thread to signal it.
- `pthread_cond_signal`:
  - Unblock at least one of the threads that are blocked on the condition variable

Condition Variables (Cont.)

- A thread calling wait routine:

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t init = PTHREAD_COND_INITIALIZER;
pthread_mutex_lock(&lock);
while (initialized == 0)
    pthread_cond_wait(&init, &lock);
pthread_mutex_unlock(&lock);
```

- The wait call releases the lock when putting said caller to sleep.
- Before returning after being woken, the wait call re-acquires the lock. (Lock must be released later)
- A thread calling signal routine:

```
pthread_mutex_lock(&lock);
initialized = 1;
pthread_cond_signal(&init);
pthread_mutex_unlock(&lock);
```

Condition Variables (Cont.)

- The waiting thread **re-checks** the condition in a while loop, instead of a simple if statement.

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t init = PTHREAD_COND_INITIALIZER;
pthread_mutex_lock(&lock);
while (initialized == 0)

    pthread_cond_wait(&init, &lock);
pthread_mutex_unlock(&lock);
```

- Sometimes a 'spurious' signal may get delivered (i.e. `pthread_cond_signal(&init)` is called but 'initialized' has not changed)
- Without rechecking, the waiting thread will continue thinking that the condition has changed *even though it has not*.

Condition Variables (Cont.)

- Don't ever to this.
  - A thread calling wait routine:

```
while(initialized == 0)
    ; // spin
```

  - A thread calling signal routine:

```
initialized = 1;
```

  - It performs poorly in many cases. □ Just wastes CPU cycles.
  - It is error prone.

Compiling and Running

- To compile them, you must include the header `pthread.h`
  - Explicitly link with the pthreads library, by adding the `-pthread` flag.

```
prompt> gcc -o main main.c -Wall -pthread
```

  - For more information,

```
man -k pthread
```