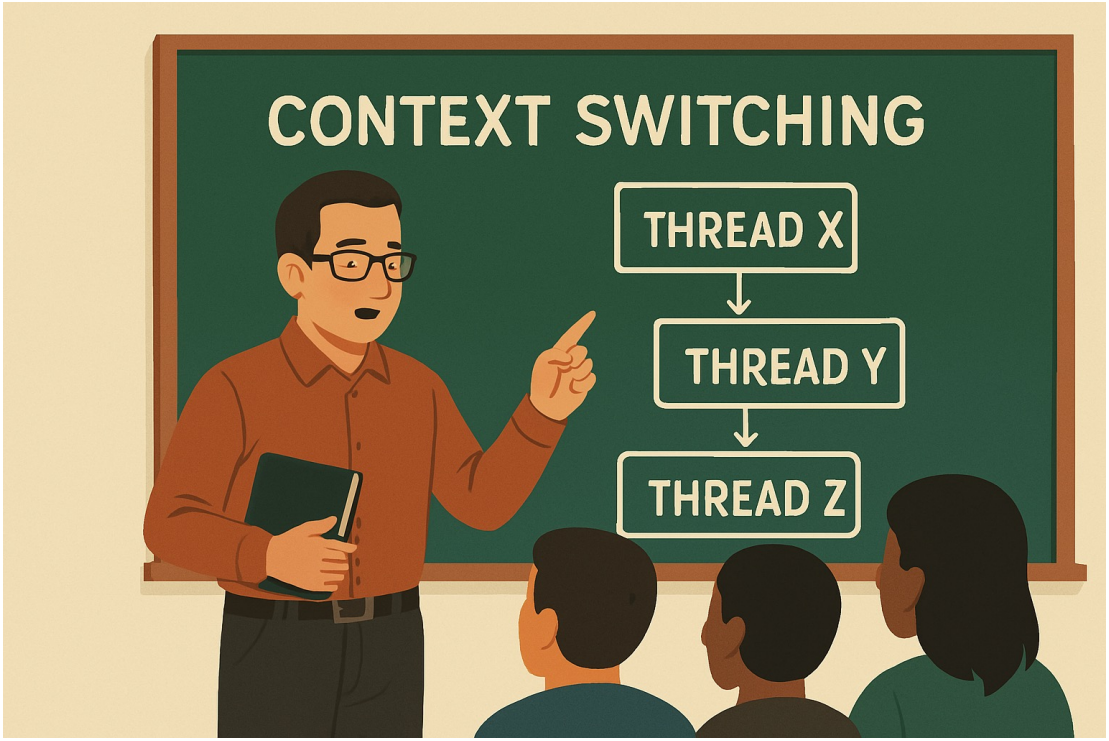
CS302

Context Switching



Spring 2025

Arkaprava Basu & Babak Falsafi

parsa.epfl.ch/course-info/cs302

Adapted from slides originally developed by Profs. Falsafi, Fatahalian, Mowry, Wenisch of CMU, Michigan

Where are We?

M	T	W	T	F
17-Feb	18-Feb	19-Feb	20-Feb	21-Feb
24-Feb	25-Feb	26-Feb	27-Feb	28-Feb
3-Mar	4-Mar	5-Mar	6-Mar	7-Mar
10-Mar	11-Mar	12-Mar	13-Mar	14-Mar
17-Mar	18-Mar	19-Mar	20-Mar	21-Mar
24-Mar	25-Mar	26-Mar	27-Mar	28-Mar
31-Mar	1-Apr	2-Apr	3-Apr	4-Apr
7-Apr	8-Apr		10-Apr	11-Apr
14-Apr	15-Apr	16-Apr	17-Apr	18-Apr
21-Apr	22-Apr	23-Apr	24-Apr	25-Apr
28-Apr	29-Apr	30-Apr	1-May	2-May
5-May	6-May	7-May	8-May	9-May
12-May	13-May	14-May	15-May	16-May
19-May	20-May	21-May	22-May	23-May
26-May	27-May	28-May	29-May	30-May

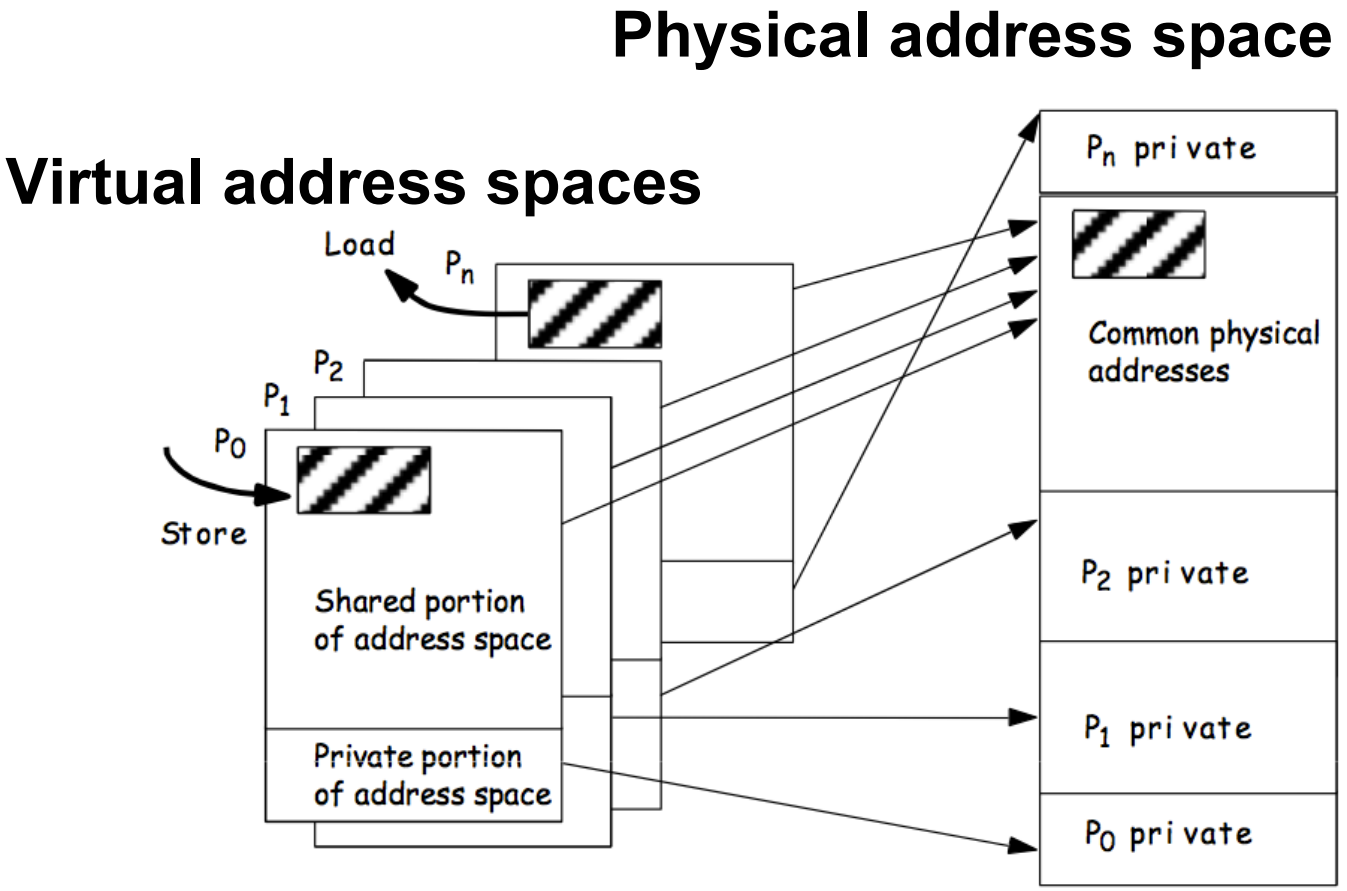
- Threads and Context Switching
- Exercise session
  - Go over midterm exam solutions
- Next Tuesday:
  - Coroutines

Heads Up

- HW 5 is now available on Moodle
  - Deadline to submit: Monday April 14th, 23:59
- HW 4 grades will be released by this week
- April 18th – April 25th is Easter Break
  - No classes during this time

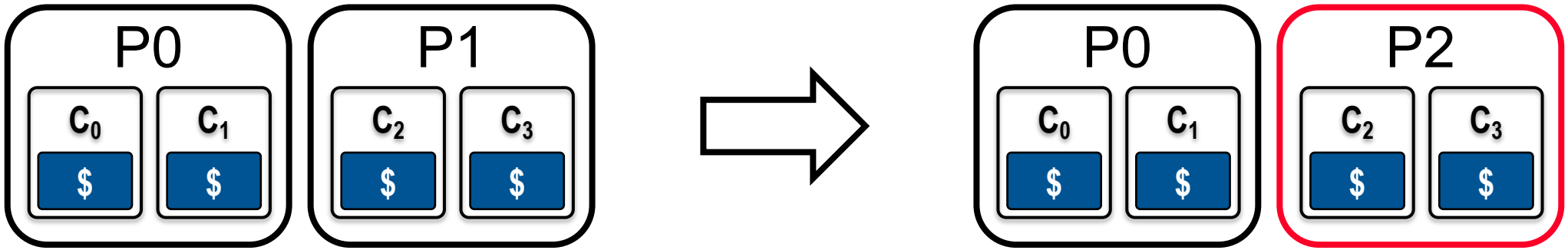
Processes

- A process is a running instance of a program
- Processes are independent of each other
- Each process has its own virtual address space
- Processes do not share any system resources



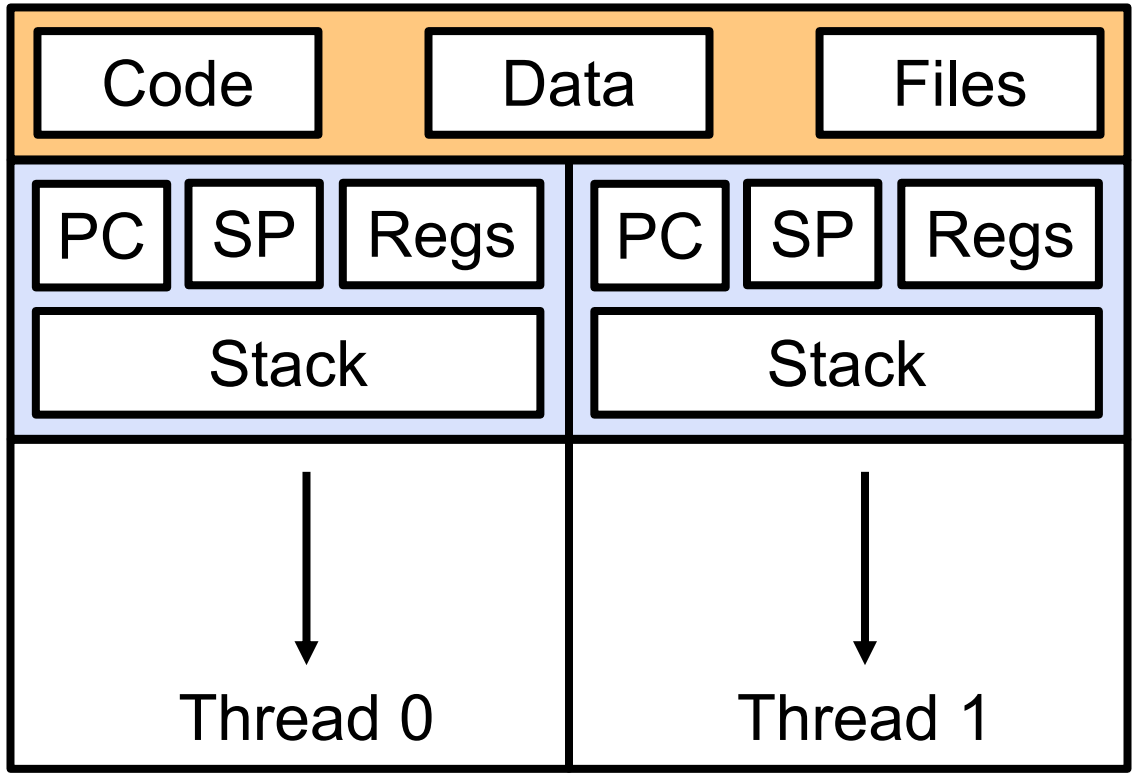
Multi-Processing

- Hundreds of processes can exist on a CPU
  - But, only a few of them run in parallel at any given time
  - Note: “P” in the pictures stands for process (not processor)
- The OS schedules and switches processes as needed
  - Gives the illusion of multi-tasking and responsiveness
  - Allows for concurrency (other process running when one is blocked)



Multi-Threading Inside Processes

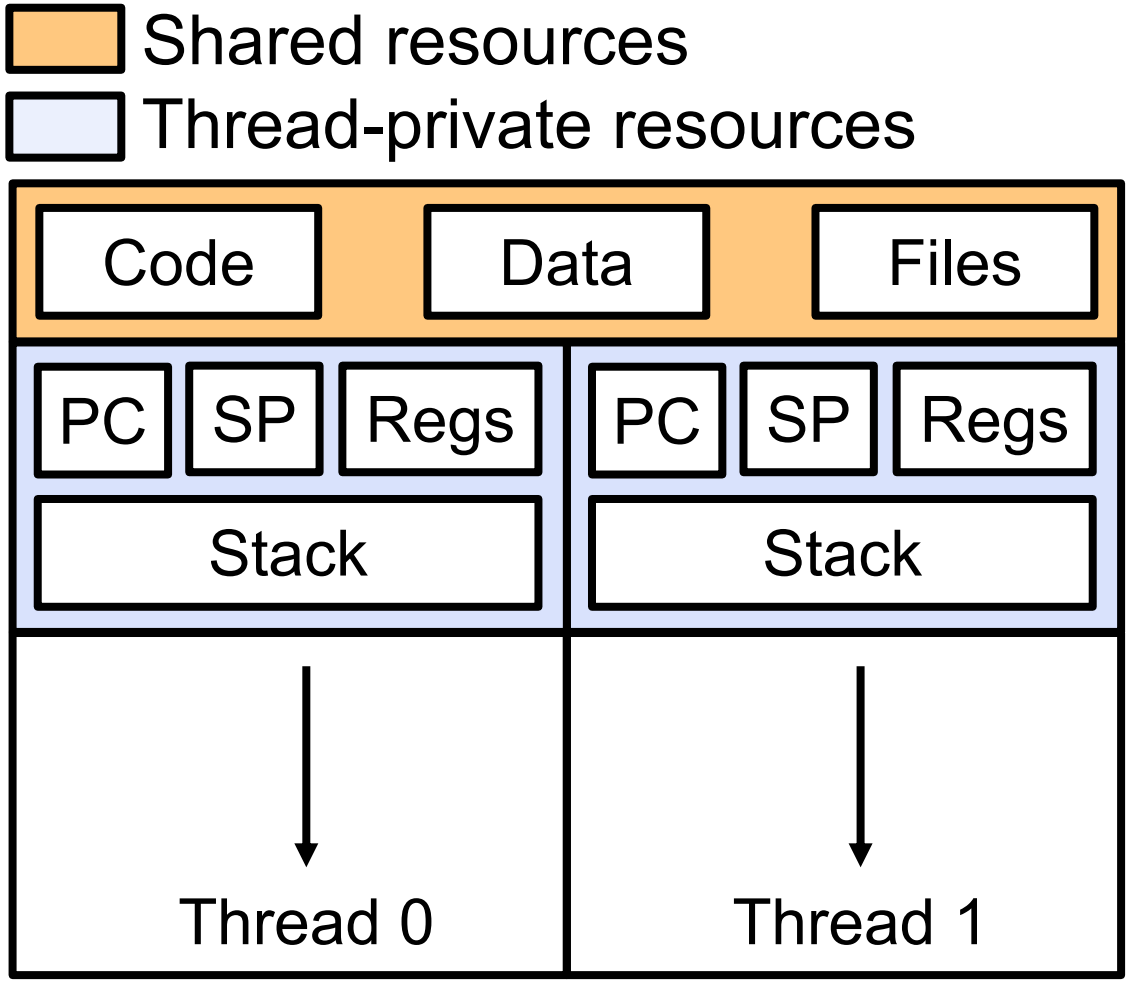
- A single process usually runs multiple threads at the same time
  - For example: OpenMP programs, Web servers, Game engines, etc.



Example process with two threads

Threads

- Threads are independent
  - Can execute various functionality
  - Each have their own program counter (PC), stack, stack pointer (SP) and variable values
- Unlike processes, threads also share resources
  - The heap, code and global data of the process is shared
  - All threads operate in the same virtual address space



Previously Seen: OpenMP Threads

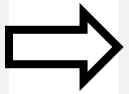
```
#pragma omp parallel for
for(int i = 0; i < N; i++)
      arr[i] = compute(i);
```

- OpenMP threads are primarily used to parallelize computation
  - Typically applied to split independent loop iterations amongst threads
  - This simple example computes array elements in parallel

- Expect to get N× speedup when running N threads on N cores

OpenMP is an Abstraction Over POSIX threads (Pthreads)

```
#pragma omp parallel for
for(int i = 0; i < N; i++)
        arr[i] = compute(i);
```



```
typedef struct {
    int start; int end;
} ThreadData;

void* worker(void* arg) {
    ThreadData* data = (ThreadData*) arg;
    for (int i = data->start; i < data->end; ++i)
        arr[i] = compute(i);
    return NULL;
}

int main() {
    pthread_t threads[NUM_THREADS];
    ThreadData thread_data[NUM_THREADS];
    int chunk_size = N / NUM_THREADS;
    for (int t = 0; t < NUM_THREADS; t++) {
        thread_data[t].start = t * chunk_size;
        thread_data[t].end = (t + 1) * chunk_size;
        pthread_create(&threads[t], NULL, worker, &thread_data[t]);
    }
    for (int t = 0; t < NUM_THREADS; t++) {
        pthread_join(threads[t], NULL);
    }
    return 0;
}
```

OpenMP is an Abstraction Over Pthreads

```
typedef struct {
    int start; int end;
} ThreadData;

void* worker(void* arg) {
    ThreadData* data = (ThreadData*) arg;
    for (int i = data->start; i < data->end; ++i)
        arr[i] = compute(i);
    return NULL;
}

int main() {
    pthread_t threads[NUM_THREADS];
    ThreadData thread_data[NUM_THREADS];
    int chunk_size = N / NUM_THREADS;
    for (int t = 0; t < NUM_THREADS; t++) {
        thread_data[t].start = t * chunk_size;
        thread_data[t].end = (t + 1) * chunk_size;
        pthread_create(&threads[t], NULL, worker, &thread_data[t]);
    }
    for (int t = 0; t < NUM_THREADS; t++) {
        pthread_join(threads[t], NULL);
    }
    return 0;
}
```

Structure to store the start and end iteration index for each thread

OpenMP is an Abstraction Over Pthreads

```
typedef struct {
    int start; int end;
} ThreadData;

void* worker(void* arg) {
    ThreadData* data = (ThreadData*) arg;
    for (int i = data->start; i < data->end; ++i)
        arr[i] = compute(i);
    return NULL;
}

int main() {
    pthread_t threads[NUM_THREADS];
    ThreadData thread_data[NUM_THREADS];
    int chunk_size = N / NUM_THREADS;
    for (int t = 0; t < NUM_THREADS; t++) {
        thread_data[t].start = t * chunk_size;
        thread_data[t].end = (t + 1) * chunk_size;
        pthread_create(&threads[t], NULL, worker, &thread_data[t]);
    }
    for (int t = 0; t < NUM_THREADS; t++) {
        pthread_join(threads[t], NULL);
    }
    return 0;
}
```

The function executed by each thread

OpenMP is an Abstraction Over Pthreads

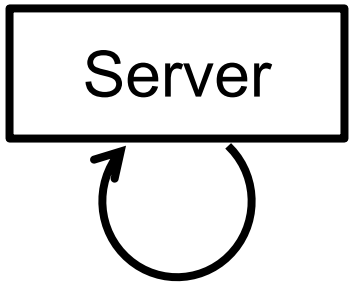
```
typedef struct {
    int start; int end;
} ThreadData;

void* worker(void* arg) {
    ThreadData* data = (ThreadData*) arg;
    for (int i = data->start; i < data->end; ++i)
        arr[i] = compute(i);
    return NULL;
}

int main() {
    pthread_t threads[NUM_THREADS];
    ThreadData thread_data[NUM_THREADS];
    int chunk_size = N / NUM_THREADS;
    for (int t = 0; t < NUM_THREADS; t++) {
        thread_data[t].start = t * chunk_size;
        thread_data[t].end = (t + 1) * chunk_size;
        pthread_create(&threads[t], NULL, worker, &thread_data[t]);
    }
    for (int t = 0; t < NUM_THREADS; t++) {
        pthread_join(threads[t], NULL);
    }
    return 0;
}
```

Create N threads and passing each thread their arguments

PThreads

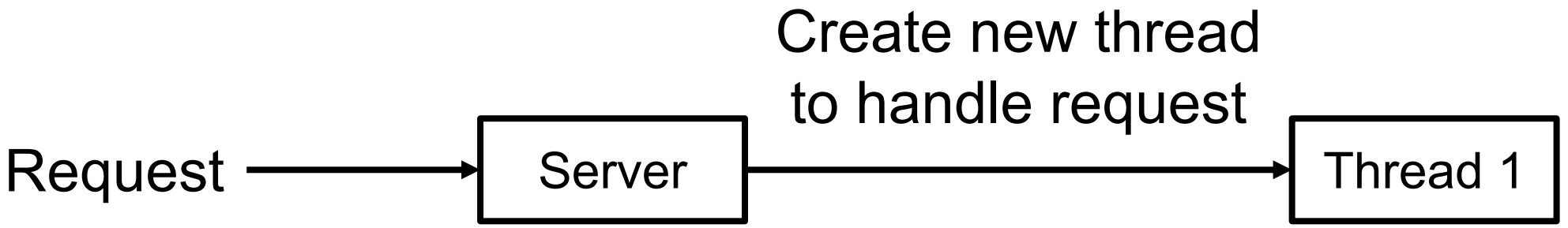
- OpenMP is simply an easy-to-use abstraction over Pthreads
- POSIX threads (Pthreads) are general-purpose threads
  - Created, scheduled and managed by the kernel
  - These are also referred to as “kernel” threads
  - Offer fine-grained control over functionality, synchronization, and scheduling
- Many languages offer high-level thread libraries using Pthreads
  - For example, **`thread`** library in C++
  - Complex multi-threaded programs can be built using Pthreads
- Threads can also switch to avoid wasting cycles (see example)

Example Web Server



Thread 0 looping and listening for incoming requests

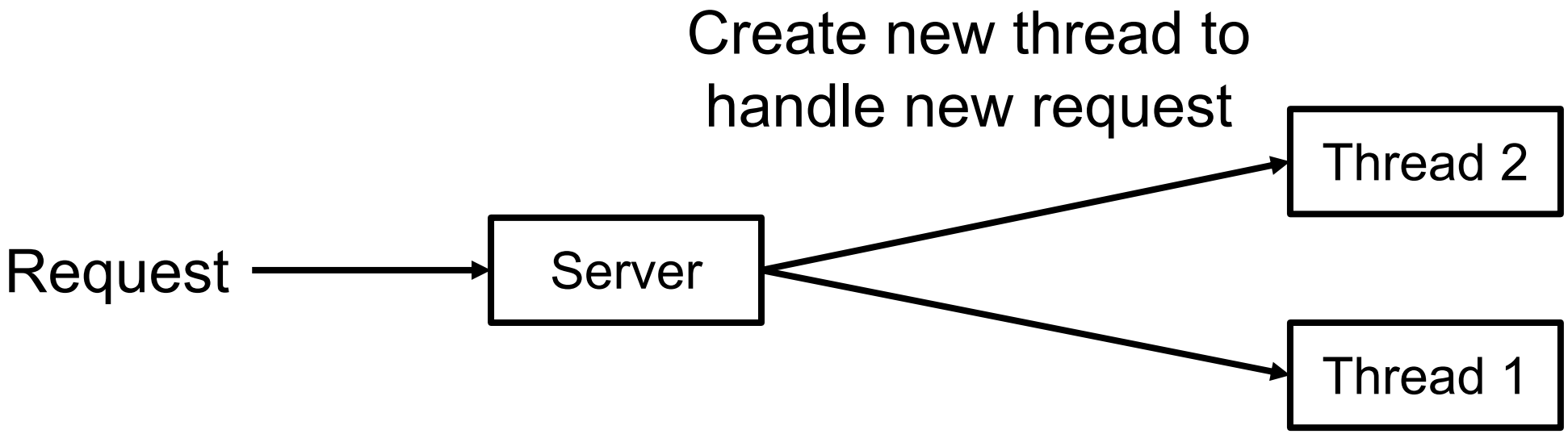
Example Web Server



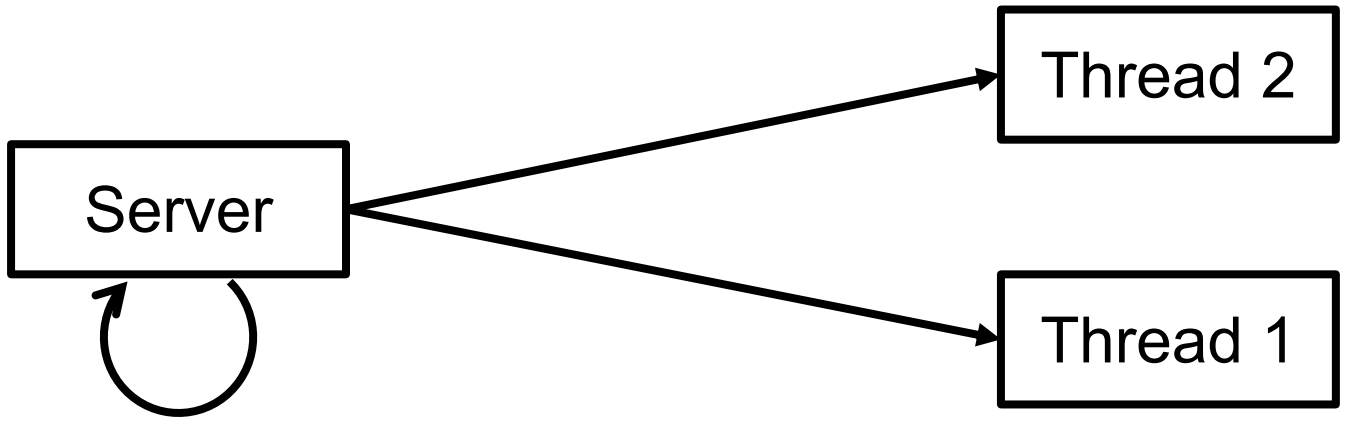
Example Web Server

Thread 0 looping and listening for incoming requests

Example Web Server



Example Web Server



Thread 0 looping and listening for incoming requests

Example Web Server

Terminate thread 1 once
finished handling the request

Thread 0 looping and listening for
incoming requests

Example Web Server in C++

```
struct Request {
    ...
};

void handle_request(Request req) {
    ...
}

int main() {
    while (true) {
        auto req = get_next_request();
        if (req) {
            std::thread t(handle_request, *req);
            t.detach();
        } else {
            std::this_thread::sleep_for(std::chrono::milliseconds(50));
        }
    }
    return 0;
}
```

Example Web Server in C++

```
struct Request {
    ...
};

void handle_request(Request req) {
    ...
}

int main() {
    while (true) {
        auto req = get_next_request();
        if (req) {
            std::thread t(handle_request, *req);
            t.detach();
        } else {
            std::this_thread::sleep_for(std::chrono::milliseconds(50));
        }
    }
    return 0;
}
```

Structure to store the payload of a request

Example Web Server in C++

```
struct Request {
    ...
};

void handle_request(Request req) {
    ...
}

int main() {
    while (true) {
        auto req = get_next_request();
        if (req) {
            std::thread t(handle_request, *req);
            t.detach();
        } else {
            std::this_thread::sleep_for(std::chrono::milliseconds(50));
        }
    }
    return 0;
}
```

Function that performs an operation on a request

Example Web Server in C++

```
struct Request {
    ...
};

void handle_request(Request req) {
    ...
}

int main() {
    while (true) {
        auto req = get_next_request();
        if (req) {
            std::thread t(handle_request, *req);
            t.detach();
        } else {
            std::this_thread::sleep_for(std::chrono::milliseconds(50));
        }
    }
    return 0;
}
```

A predefined function that gets the next request from a queue

Example Web Server in C++

```
struct Request {
    ...
};

void handle_request(Request req) {
    ...
}

int main() {
    while (true) {
        auto req = get_next_request();
        if (req) {
            std::thread t(handle_request, *req);
            t.detach();
        } else {
            std::this_thread::sleep_for(std::chrono::milliseconds(50));
        }
    }
    return 0;
}
```

If not NULL, then spawns a thread that executes **`handle_request`** for **`*req`**

Example Web Server in C++

```
struct Request {
    ...
};

void handle_request(Request req) {
    ...
}

int main() {
    while (true) {
        auto req = get_next_request();
        if (req) {
            std::thread t(handle_request, *req);
            t.detach();
        } else {
            std::this_thread::sleep_for(std::chrono::milliseconds(50));
        }
    }
    return 0;
}
```

Makes sure that once thread finishes executing the function, it exits

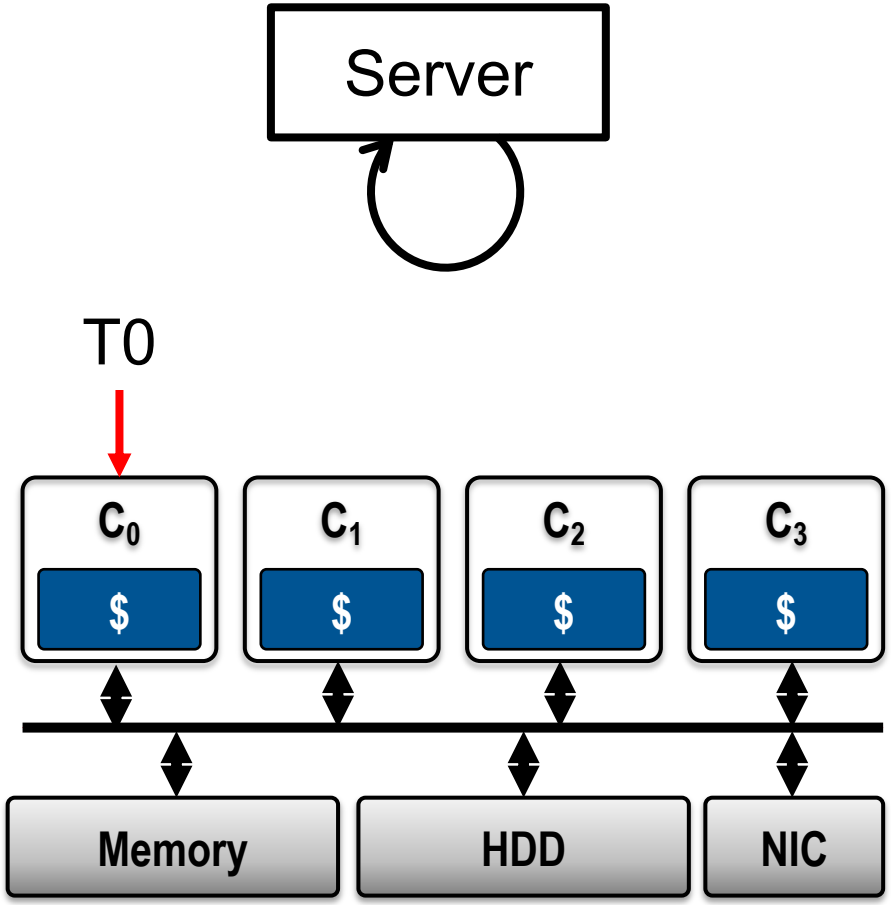
Example Web Server in C++

```
struct Request {
    ...
};

void handle_request(Request req) {
    ...
}

int main() {
    while (true) {
        auto req = get_next_request();
        if (req) {
            std::thread t(handle_request, *req);
            t.detach();
        } else {
            std::this_thread::sleep_for(std::chrono::milliseconds(50));
        }
    }
    return 0;
}
```

If no request found, then waits 50ms before checking again

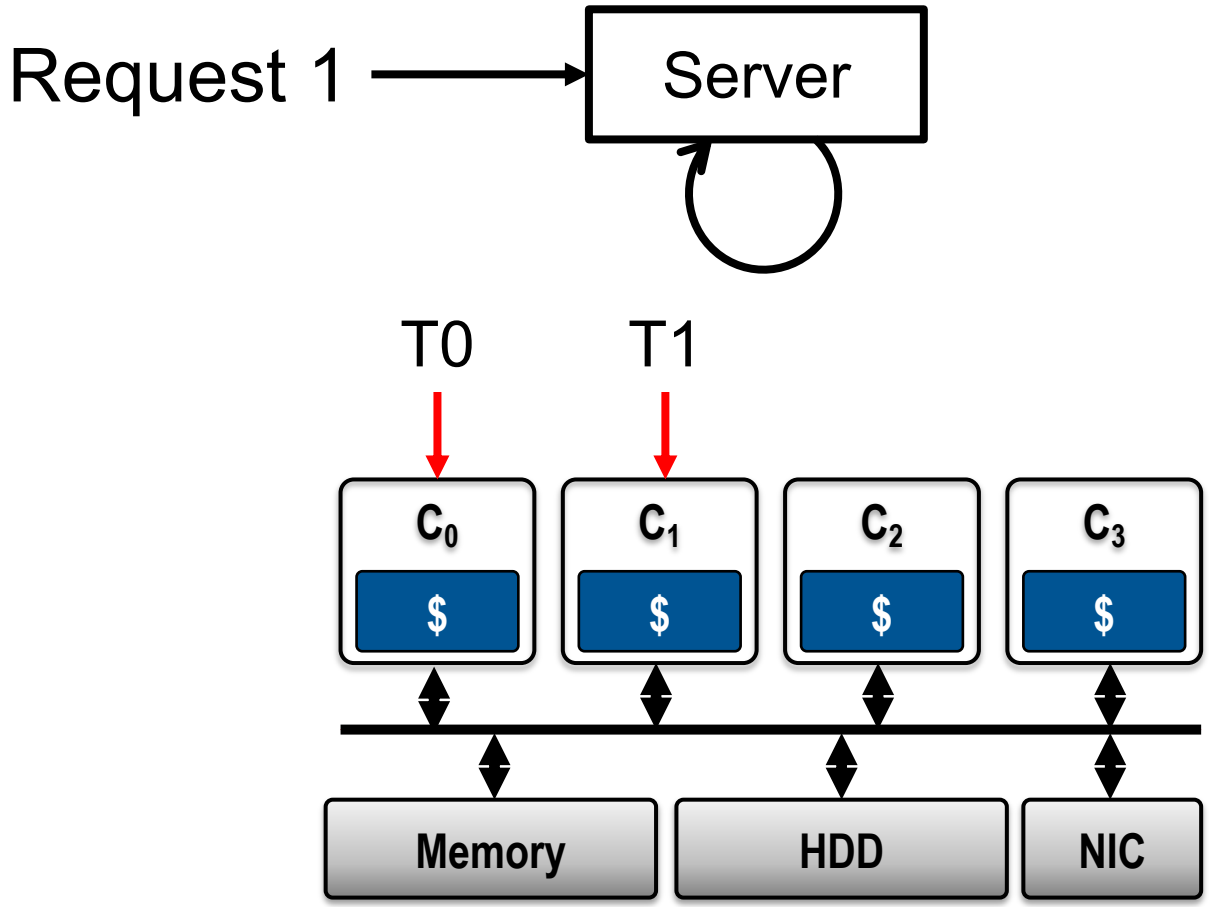
Running The Web Server

- Assume the web server is running on a CPU with four cores



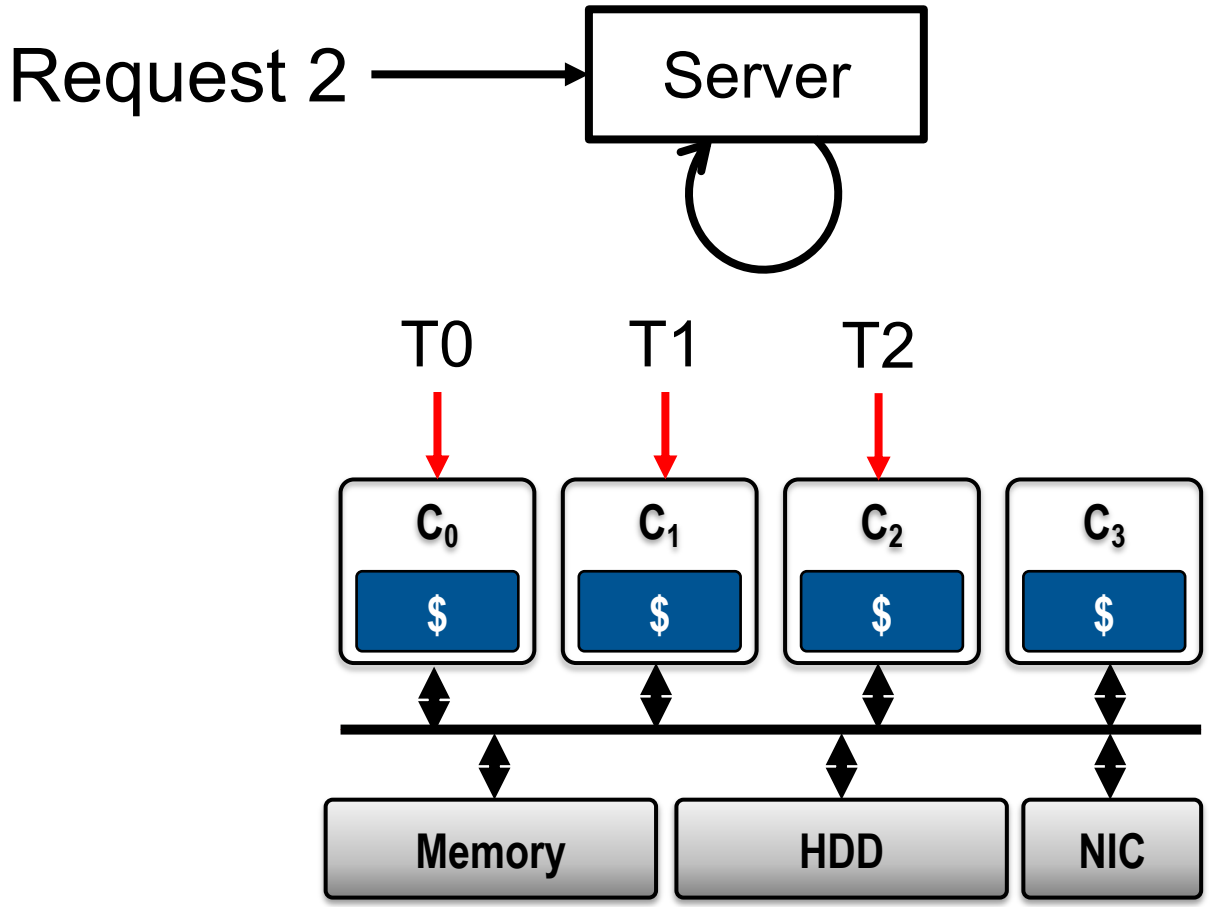
Running The Web Server

- Assume the web server is running on a CPU with four cores



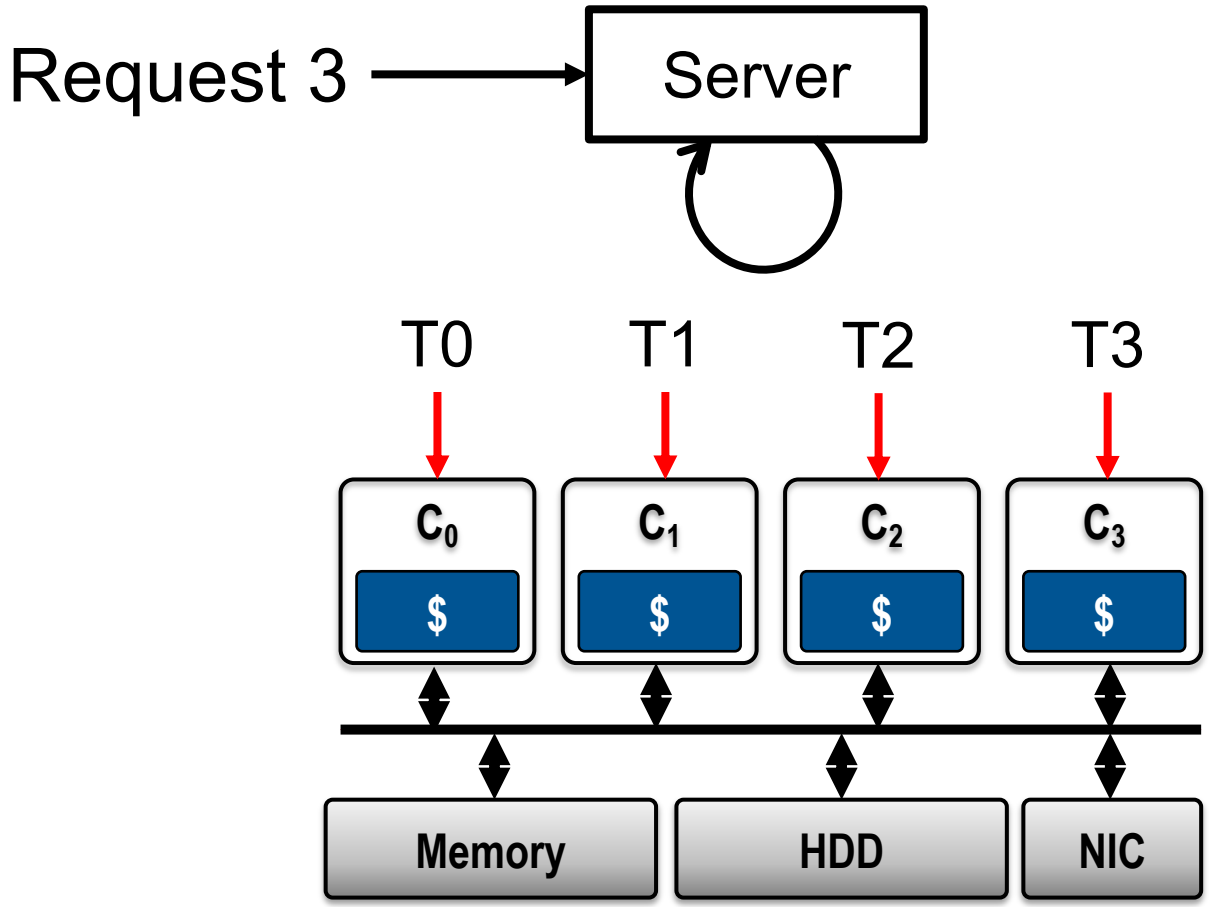
Running The Web Server

- Assume the web server is running on a CPU with four cores



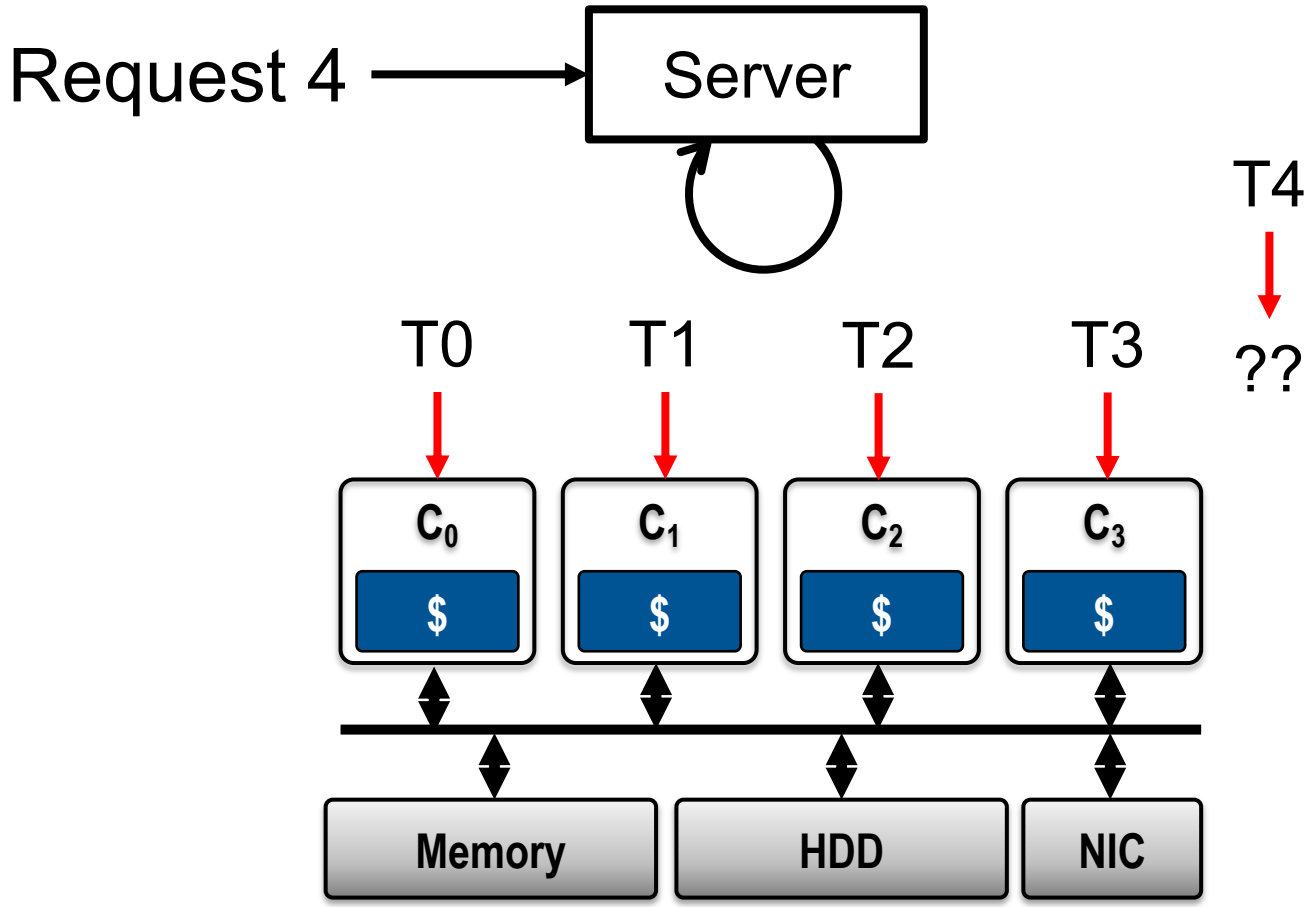
Running The Web Server

- Assume the web server is running on a CPU with four cores



Running The Web Server

- Assume the web server is running on a CPU with four cores



Running The Web Server

- At any given time, only three requests can be handled in parallel
- If there is a high rate of incoming requests,
  - The web server will spawn new threads to handle the new requests
  - These new threads will be blocked behind the threads handling the old requests
- Two options:
  - Wait for the threads executing the old requests to finish first
  - Context switch threads!

Drawback of Waiting: Requests Can Waste Cycles

```
struct Request {
    int id, type;
    std::string payload;
};

int hash_arr[N];
void handle_request(Request req) {
    if (req.type == 0) {
        std::ofstream outfile("output.txt", std::ios::app);
        outfile << "Request " << req.id << ": " << req.payload << "\n";
        outfile.close();
    } else if (req.type == 1) {
        hash_arr[req.id] = calc_hash(req.payload);
    }
}
```

Drawback of Waiting: Requests Can Waste Cycles

```
struct Request {
    int id, type;
    std::string payload;
};

int hash_arr[N];
void handle_request(Request req) {
    if (req.type == 0) {
        std::ofstream outfile("output.txt", std::ios::app);
        outfile << "Request " << req.id << ": " << req.payload << "\n";
        outfile.close();
    } else if (req.type == 1) {
        hash_arr[req.id] = calc_hash(req.payload);
    }
}
```

There are two types of requests

Each request contains a string payload

Drawback of Waiting: Requests Can Waste Cycles

```
struct Request {
    int id, type;
    std::string payload;
};

int hash_arr[N];
void handle_request(Request req) {
    if (req.type == 0) {
        std::ofstream outfile("output.txt", std::ios::app);
        outfile << "Request " << req.id << ": " << req.payload << "\n";
        outfile.close();
    } else if (req.type == 1) {
        hash_arr[req.id] = calc_hash(req.payload);
    }
}
```

Global array to store SHA hash of the payloads for each request

Drawback of Waiting: Requests Can Waste Cycles

```
struct Request {
    int id, type;
    std::string payload;
};

int hash_arr[N];
void handle_request(Request req) {
    if (req.type == 0) {
        std::ofstream outfile("output.txt", std::ios::app);
        outfile << "Request " << req.id << ": " << req.payload << "\n";
        outfile.close();
    } else if (req.type == 1) {
        hash_arr[req.id] = calc_hash(req.payload);
    }
}
```

Type 0 requests write the string into a file (I/O operation)

Drawback of Waiting: Requests Can Waste Cycles

```
struct Request {
    int id, type;
    std::string payload;
};

int hash_arr[N];
void handle_request(Request req) {
    if (req.type == 0) {
        std::ofstream outfile("output.txt", std::ios::app);
        outfile << "Request " << req.id << ": " << req.payload << "\n";
        outfile.close();
    } else if (req.type == 1) {
        hash_arr[req.id] = calc_hash(req.payload);
    }
}
```

Type 1 requests compute the SHA hash of the string and store it

Drawback of Waiting: Requests Can Waste Cycles

```
// Thread handling a type 0 request

std::ofstream outfile("output.txt", std::ios::app);
outfile << "Request " << req.id << ": " << req.payload << "\n";
outfile.close();
```

- Writing a single line to a file can take between 100s µs to ms
  - Flash/HDD raw access time and syscall overhead
- So, a thread handling a type 0 request is simply idle during I/O
  - > 100 µs are wasted per line

Wasted Cycles Could Be Harnessed

```
// Thread handling a type 1 request

hash_arr[req.id] = calc_hash(req.payload);
```

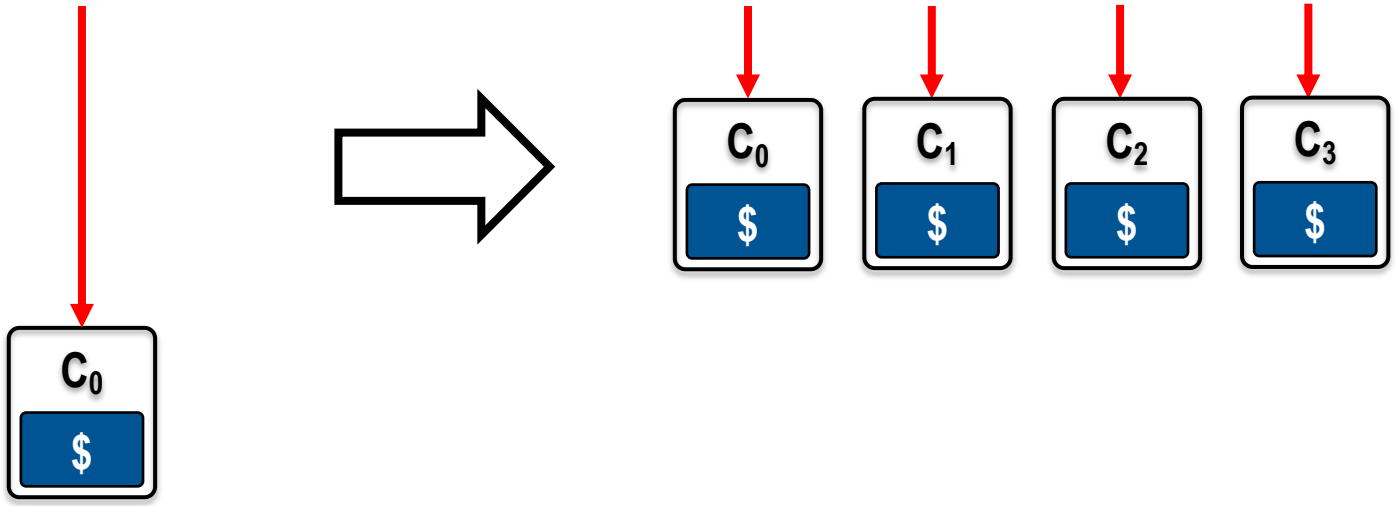
- The cycles wasted by type 0 requests could be harnessed
  - Computing the SHA hash takes 1-10 µs
  - Ten type 1 requests could be serviced in the wasted time per line

- Threads executing type 1 requests can get blocked behind threads executing type 0 requests that are wasting clock cycles!

Concurrency vs. Parallelism

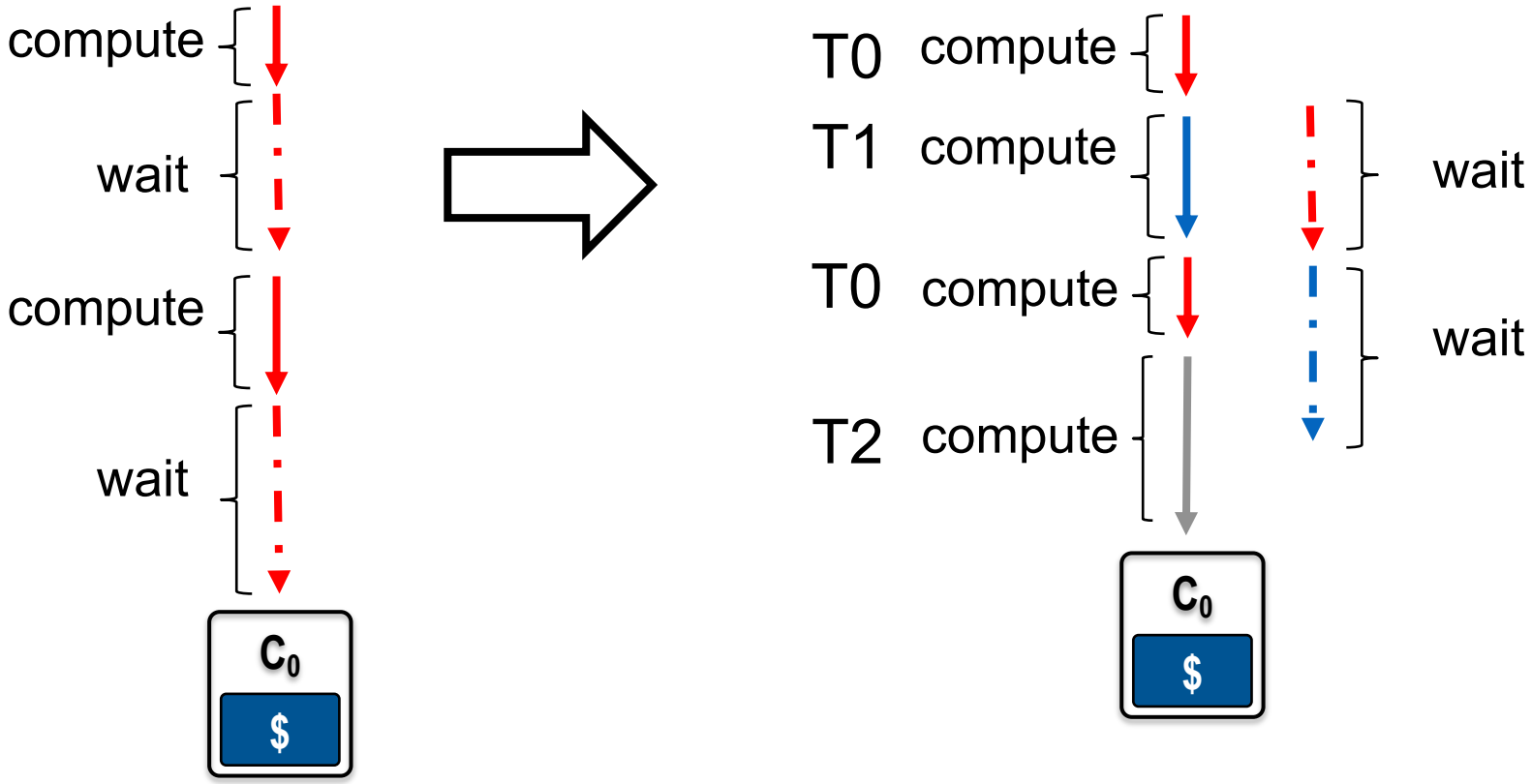
- Parallelism is the ability to execute multiple threads at the same time
  - Speed up a single task by splitting it amongst multiple threads

- Parallelism exploits multiple cores



Concurrency vs. Parallelism

- Concurrency is the ability to hide latency among blocked threads
  - Increase utilization of a core and throughput



Context Switching Threads

```
// Thread handling a type 0 request

std::ofstream outfile("output.txt", std::ios::app);
outfile << "Request " << req.id << ": " << req.payload << "\n";
outfile.close();
```

- Reminder: Pthreads are “kernel” threads
  - The kernel has full control over how to schedule the threads
- The highlighted line results in syscalls for the I/O operation
  - The kernel switches the thread with a “ready” thread
  - Once I/O operation is done, the kernel switches back the waiting thread

The Kernel Co-ordinates the Switch

- The kernel co-ordinates the switch:
  - Saves the context of the current thread
  - Calls the scheduler algorithm to pick the next thread
  - Restores the context of the next thread
- The next thread resumes execution
- Question:
  - What context needs to be saved so that threads can be paused and restored without any data loss?

Example of Context

- The “state” of a thread that must be saved before a switch
- Consider the following snippet of assembly code:

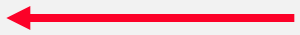
```
lw r1, 0(r2)
lw r6, 0(r3)

/* Switch to a different thread */
...
/* Resume normal execution */

mul r1, r1, r4
add r6, r1, r6
sw r6, 0(r3)
```

Assume switch happens here

Example of Context

- The “state” of a thread that must be saved before a switch
- Consider the following snippet of assembly code:

```
lw r1, 0(r2)
lw r6, 0(r3)

/* Switch to a different thread */
...
/* Resume normal execution */

mul r1, r1, r4
add r6, r1, r6
sw r6, 0(r3)
```

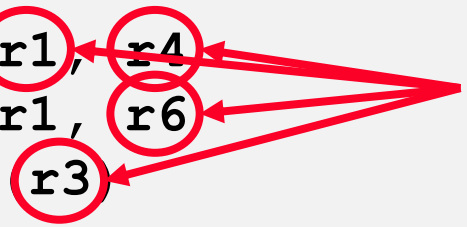


Need to remember the PC of the last executed instruction

Example of Context

- The “state” of a thread that must be saved before a switch
- Consider the following snippet of assembly code:

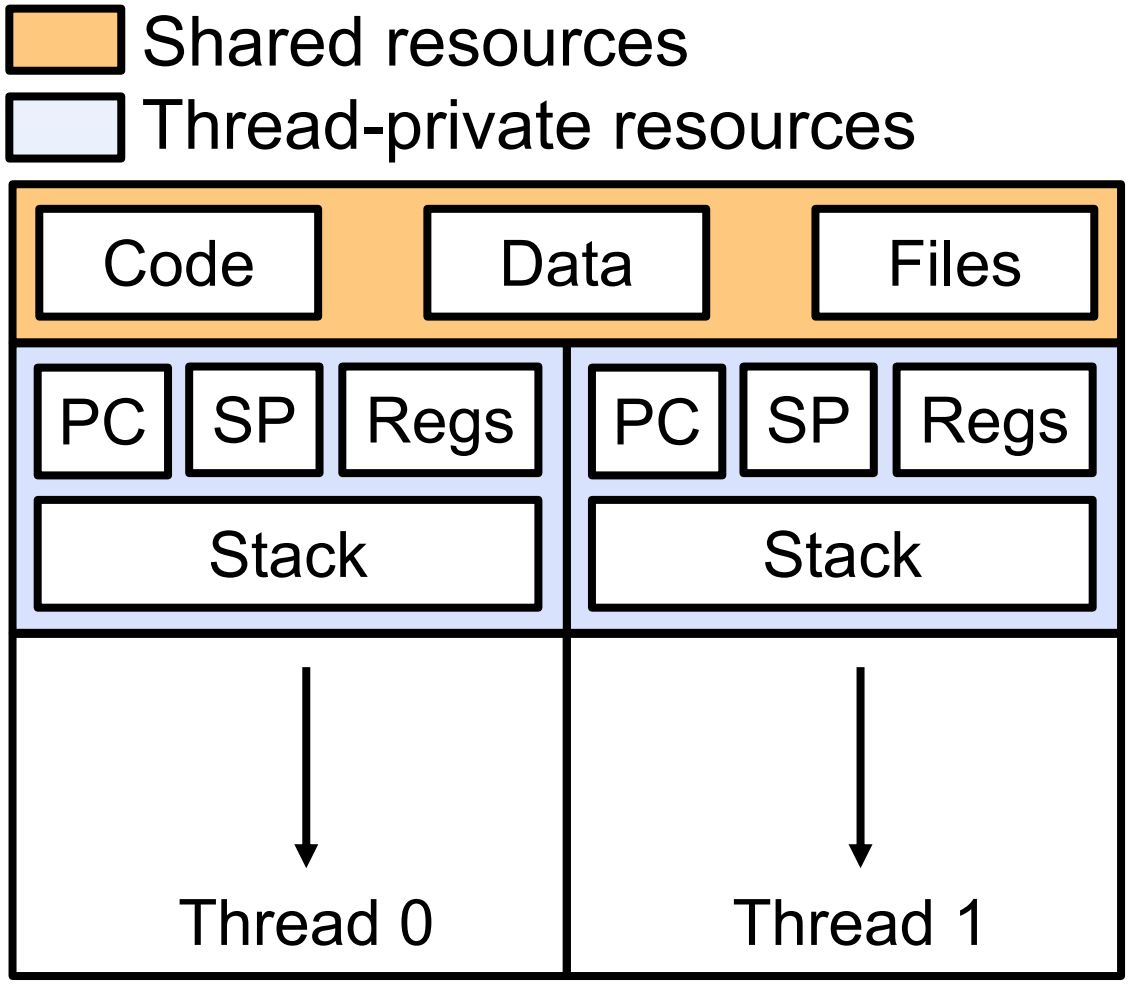
```
lw r1, 0(r2)
lw r6, 0(r3)

/* Switch to a different thread */
...
/* Resume normal execution */

mul r1, r1, r4
add r6, r1, r6
sw r6, 0(r3)
```



Need to remember the value of all registers before the switch

Thread Context

- On a switch, a thread will be replaced by another thread from the same process
- No need to save shared resources on a switch, e.g.,
  - Heap, Code and Data segments, File descriptors, Process attributes, etc.
- Only save thread-private:
  - Register values, Stack, PC and SP



Thread Switching Overhead

- Explicit overhead:
  - Syscall means let the pipeline empty, use the trap table to jump to the syscall code, switch to OS
  - Dump the current threads' context to memory, load the new thread's context
  - Return from syscall, switch back to user

- Implicit overhead:
  - All microarchitectural state that is shared (branch tables, TLB, cache hierarchy) is affected while a thread is running
  - The next thread may not find all it state left behind

Is Context Switching Worth It?

- A single context switch between threads in Linux takes ~1 - 5μs
- Switching is only worth it if the time to switch → run → switch back is less than idle time (while waiting)

- In our example,
  - Context switch time is a few μs ~ Type 1 request execution time
  - Would be better if context switching took less time

- Context switching overhead increases with the number of threads
  - The kernel needs to keep track of more threads

Summary

- Threads are independent units of execution within a process
- Pthreads are general purpose kernel threads
  - Enable concurrency, not just parallelism
- If a thread blocks, the core sits idle unless another thread can run
- Context switching lets the core switch between threads
  - Comes with overhead (~1–5 µs for kernel threads)