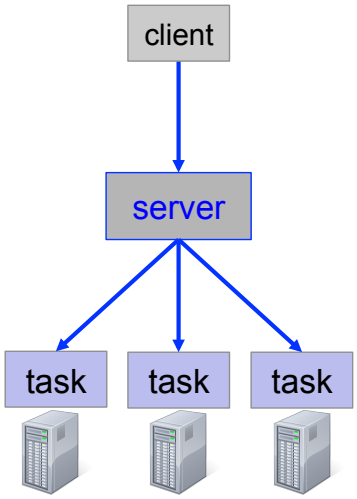
Concurrency and Threading: Introduction

- Concurrency: "Why?"
- Concurrency: "How?"
  - Serial, Collaborative, and Preemptive Task Management
- Concurrent Tasks and Shared Global State
- Preemptive Task Management: Threads
- Example: POSIX pthreads

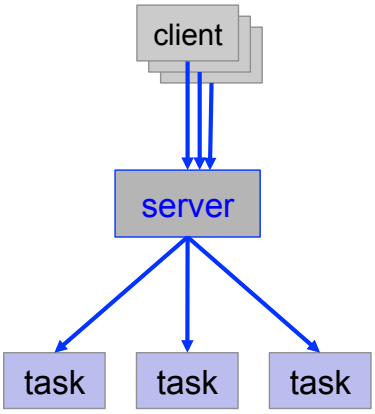
Concurrency: "Why?"

1. Latency Reduction:

- Apply parallel algorithm.
- Concurrency in trivially parallelizable problems.



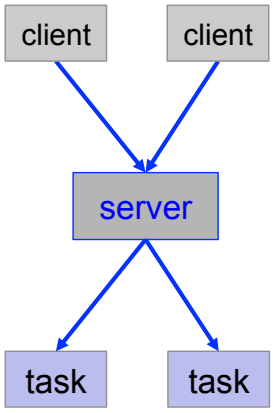
Concurrency: "Why?"



1. Latency Reduction:
- Apply parallel algorithm.
- Concurrency in trivially parallelizable problems.

2. Latency Hiding:
- Use concurrency to perform useful work while another operation is pending. (multiprogramming)
- Latency of operation is not affected, but hidden.

Concurrency: "Why?"

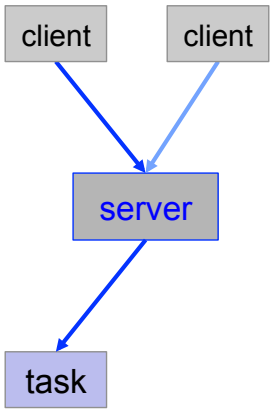


1. Latency Reduction:
- Apply parallel algorithm.
- Concurrency in trivially parallelizable problems.

2. Latency Hiding:
- Use concurrency to perform useful work while another operation is pending. (multiprogramming)
- Latency of operation is not affected, but hidden.

3. Throughput Increase:
- Employ multiple concurrent executions of sequential threads to accommodate more simultaneous work.
- Concurrency is then handled by specialized subsystems (OS, database, etc.)

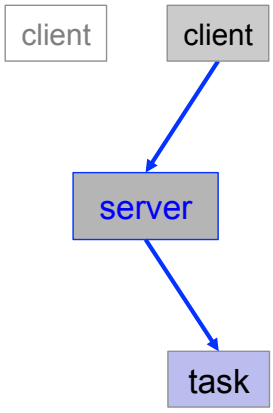
Concurrency: "How?"



Definition [**Task**]: Control flow.

- **Serial** Task Management:
  - Execute each task to completion before starting new task.

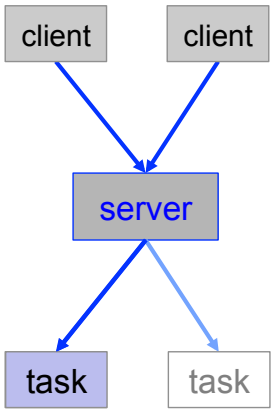
Concurrency: "How?"



Definition [**Task**]: Control flow.

- **Serial** Task Management:
  - Execute each task to completion before starting new task.

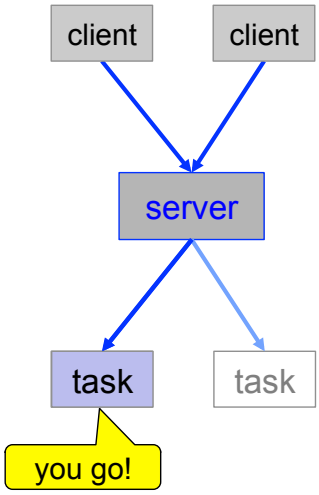
Concurrency: "How?"



Definition [**Task**]: Control flow.

- **Serial** Task Management:
  - Execute each task to completion before starting new task.
- **Cooperative** Task Management:
  - (Voluntarily) yield CPU at well-defined points in execution.

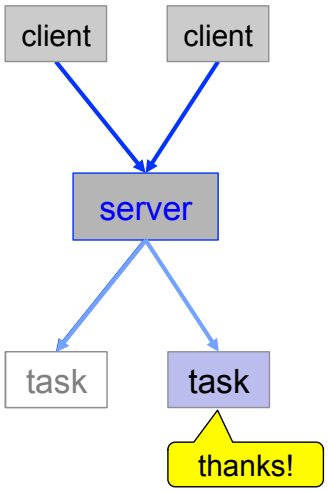
Concurrency: "How?"



Definition [**Task**]: Control flow.

- **Serial** Task Management:
  - Execute each task to completion before starting new task.
- **Cooperative** Task Management:
  - (Voluntarily) yield CPU at well-defined points in execution.

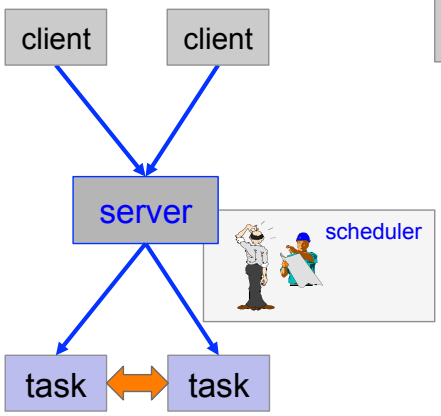
Concurrency: "How?"



Definition [**Task**]: Control flow.

- **Serial** Task Management:
  - Execute each task to completion before starting new task.
- **Cooperative** Task Management:
  - (Voluntarily) yield CPU at well-defined points in execution.

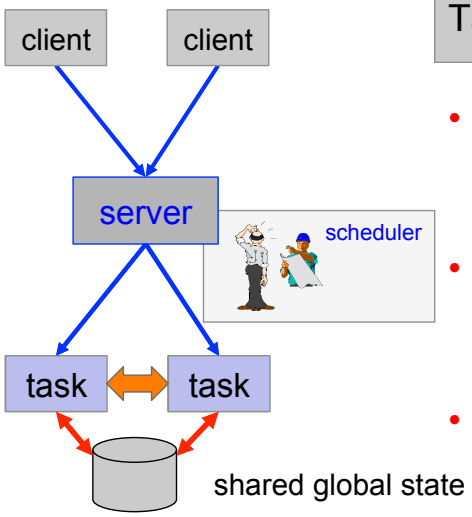
Concurrency: "How?"



Definition [**Task**]: Control flow.

- **Serial** Task Management:
  - Execute each task to completion before starting new task.
- **Cooperative** Task Management:
  - (Voluntarily) yield CPU at well-defined points in execution.
- **Preemptive** Task Management:
  - Execution of tasks can interleave.

Concurrency: "How?"



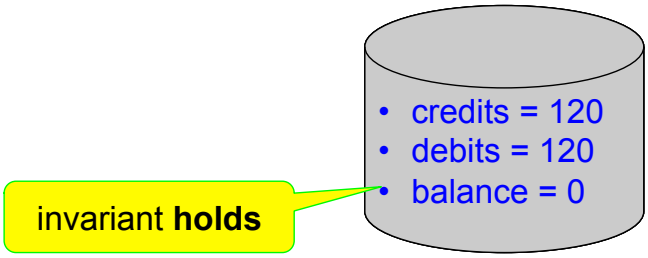
Definition [**Task**]: Control flow.

Tasks have access to **shared global state.**

- **Serial** Task Management:
  - Execute each task to completion before starting new task.
- **Cooperative** Task Management:
  - (Voluntarily) yield CPU at well-defined points in execution.
- **Preemptive** Task Management:
  - Execution of tasks can interleave.

Shared Global State

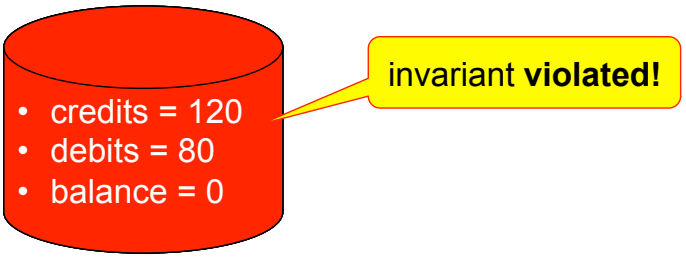
Example – Invariant: credits - debits == balance



Definition [**Invariant**]: Predicate that "has to be true at all times".

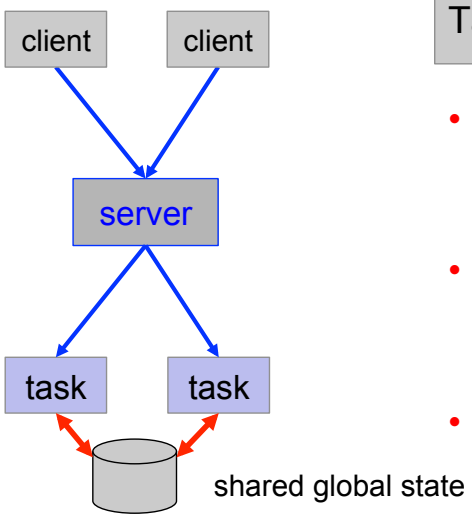
Shared Global State

Example – Invariant: credits - debits == balance



Definition [**Invariant**]: Predicate that "has to be true at all times".

Concurrency: "How?"

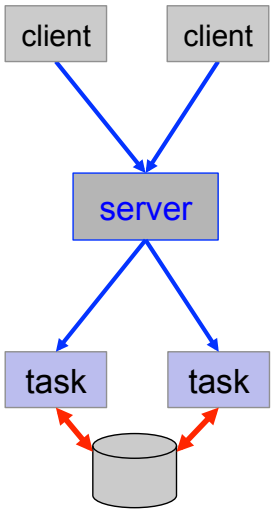


Definition [**Task**]: Control flow.

Tasks have access **to shared global state**.

- **Serial** Task Management:
  - Execute each task to completion before starting new task.
- **Cooperative** Task Management:
  - (Voluntarily) yield CPU at well-defined points in execution.
- **Preemptive** Task Management:
  - Execution of tasks can interleave.

Serial Task Management



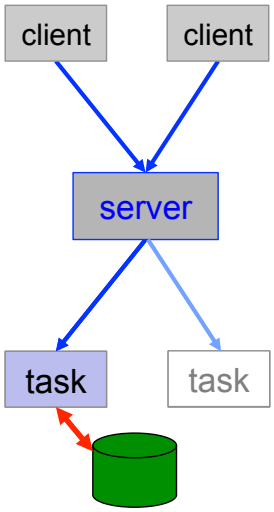
Pros:

- Only one task is running at a given time.
- No potential for conflict in accessing shared state.
- We can define so-called "**inter-task invariants**"; while one task is running, no other task can violate these invariants.

Cons:

- Only one task is running at a given time!
  - No multiprogramming.
  - No multiprocessor parallelism.

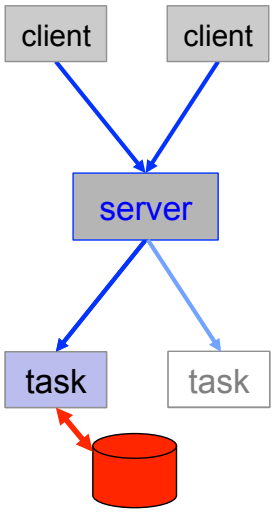
Cooperative Task Management



Pros:

- Allows for some controlled multiprogramming.

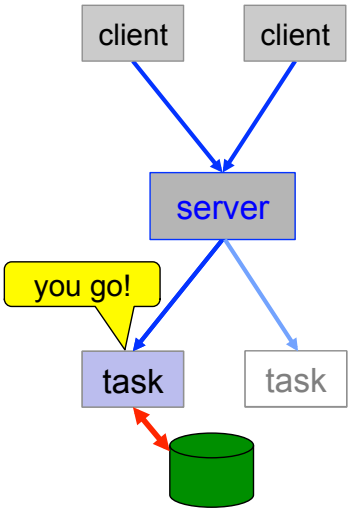
Cooperative Task Management



Pros:

- Allows for some controlled multiprogramming.

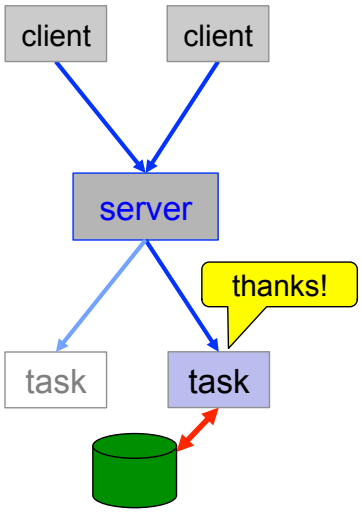
Cooperative Task Management



Pros:

- Allows for some controlled multiprogramming.
- Invariants must be ensured at yielding points only.

Cooperative Task Management



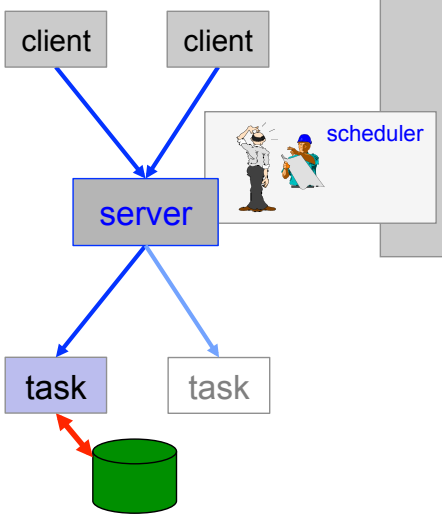
Pros:

- Allows for some controlled multiprogramming.
- Invariants must be ensured at yielding points only.

Cons:

- Invariants are not automatically enforced.

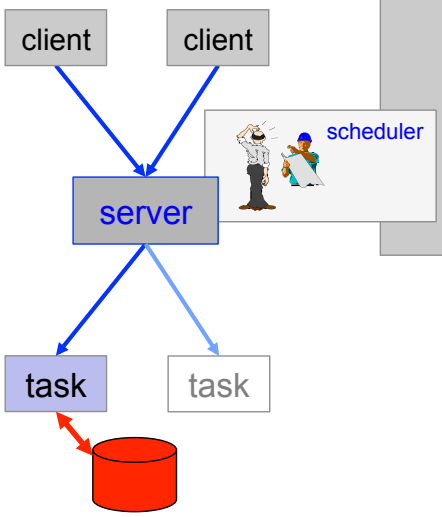
Preemptive Task Management



Pros:

- Allows for high level of multiprogramming.
- Parallelism

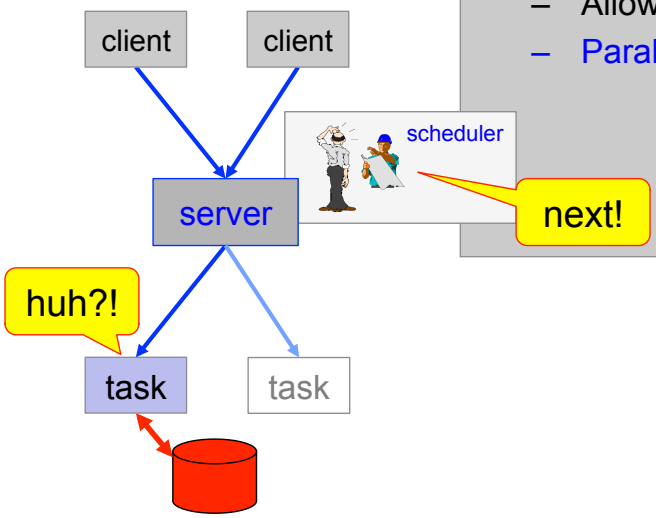
Preemptive Task Management



Pros:

- Allows for high level of multiprogramming.
- Parallelism

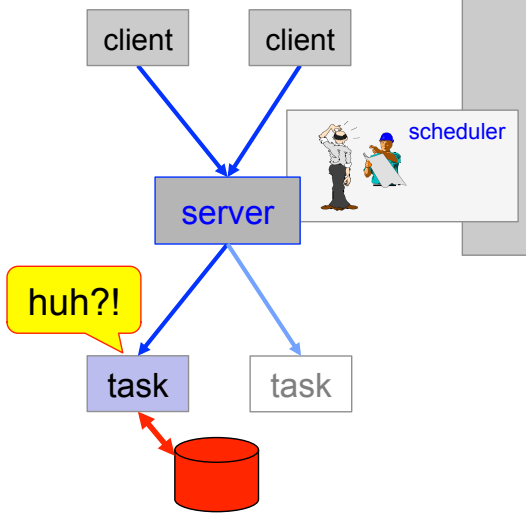
Preemptive Task Management



Pros:

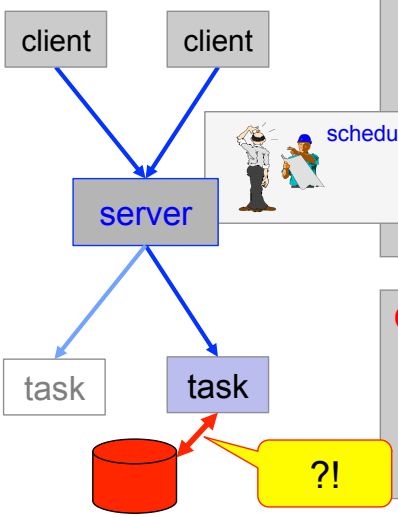
- Allows for high level of multiprogramming.
- Parallelism

Preemptive Task Management



Pros:
- Allows for high level of multiprogramming.
- Parallelism

Preemptive Task Management

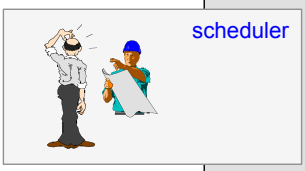


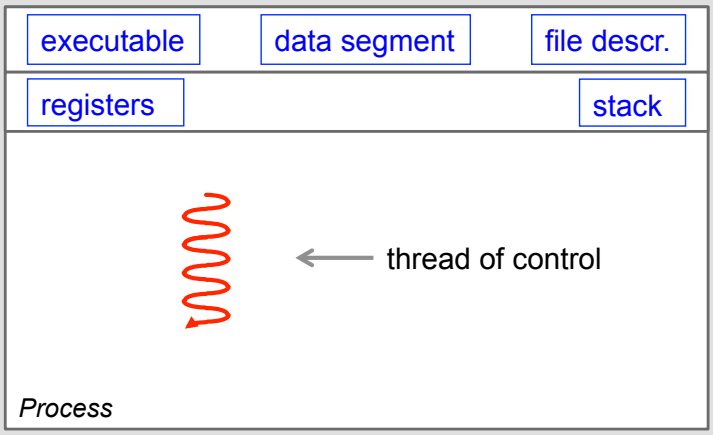
Pros:
- Allows for high level of multiprogramming.
- Parallelism

Cons:
- Invariants are must hold at all times (!?)
- Mechanism: synchronization

Preemptive Task Management: "How?"

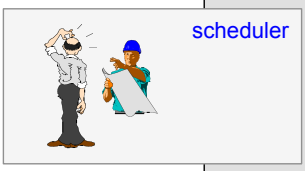
Scheduler manages **threads**

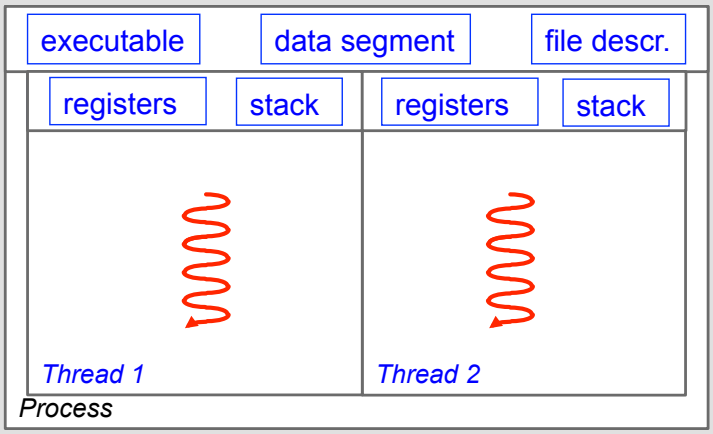




Preemptive Task Management: "How?"

Scheduler manages **threads**





Example: POSIX pthreads

- **pthread_create** (create a thread)
- **pthread_cancel** (terminate another thread)
- **pthread_detach** (have thread release res's)
- **pthread_equal** (two thread id's equal?)
- **pthread_exit** (exit a thread)
- **pthread_kill** (send a signal to a thread)
- **pthread_join** (wait for a thread)
- **pthread_self** (what is my id?)

Example: POSIX pthreads

- **pthread_create** (create a thread)
- **pthread**
- **pthread**
- **pthread**
- **pthread**
- **pthread**
- **pthread_join** (wait for a thread)

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

```
fd = open("my.dat", O_RDONLY);

if (error = pthread_create(&t_id, NULL, processfd, &fd))
   fprintf(stderr, "Failed create thread: %s\n", strerror(error));
```

pthreads: Thread Attributes

attribute objects	`pthread_attr_destroy` `pthread_attr_init`
state	`pthread_attr_getdetachstate` `pthread_attr_setdetachstate`

stack	`pthread_attr_getguardsize`
	`pthread_attr_setguardsize`
	`pthread_attr_getstack`
	`pthread_attr_setstack`

scheduling	`pthread_attr_getinheritedsched`
	`pthread_attr_setinheritedsched`
	`pthread_attr_getschedparam`
	`pthread_attr_setschedparam`
	`pthread_attr_getschedpolicy`
	`pthread_attr_setschedpolicy`
	`pthread_attr_getscope`
	`pthread_attr_setscope`

pthreads: Thread Attributes – State

attribute objects	`pthread_attr_destroy` `pthread_attr_init`
state	`pthread_attr_getdetachstate` `pthread_attr_setdetachstate`

stack	`pthread_attr_getguardsize`
	`pthread_attr_setguardsi`
	`pthread_attr_getstack`
	`pthread_attr_setstack`

scheduling	`pthread_attr_getinherit`
	`pthread_attr_setinherit`
	`pthread_attr_getschedpa`
	`pthread_attr_setschedparam`
	`pthread_attr_getschedpolicy`
	`pthread_attr_setschedpolicy`
	`pthread_attr_getscope`
	`pthread_attr_setscope`

- **Detached** threads release resources when terminate.
- **Attached** states hold on to resources until parent thread calls pthread_join.

pthreads: Thread Attributes – Stack

attribute objects	`pthread_attr_destroy` `pthread_attr_init`
state	`pthread_attr_getdetachstate` `pthread_attr_setdetachstate`

stack	`pthread_attr_getguardsize`
	`pthread_attr_setguardsize`
	`pthread_attr_getstack`
	`pthread_attr_setstack`

scheduling	`pthread_attr_getinh`
	`pthread_attr_setinh`
	`pthread_attr_getsch`
	`pthread_attr_setsch`
	`pthread_attr_getsch`
	`pthread_attr_setsch`
	`pthread_attr_getscc`
	`pthread_attr_setscc`

- **setstack** defines location and size of stack.
- **setguardsize** allocates additional memory. If the thread overflows into this extra memory, an error is generated.

pthreads: Thread Attributes – Scheduling

attribute objects	
state	
stack	

scheduling	`pthread_attr_setinheritedsched`
	`pthread_attr_getschedparam`
	`pthread_attr_setschedparam`
	`pthread_attr_getschedpolicy`
	`pthread_attr_setschedpolicy`
	`pthread_attr_getscope`
	`pthread_attr_setscope`

- **PTHREAD_INHERIT_SCHED** defines that scheduling parameters are inherited from parent thread. (as opposed to **PTHREAD_EXPLICIT_SCHED**).
- **Scheduling policies**: SCHED_FIFO, SCHED_RR, SCHED_SPORADIC, SCHED_OTHER, …
- **contention scope** defines whether process competes at process level or at system level for resources.

Concurrency and Threading: Introduction

- Concurrency: "Why?"
- Concurrency: "How?"
  - Serial Task Management
  - Collaborative Task Management
  - Preemptive Task Management
- Concurrent Tasks and Shared Global State
- Preemptive Task Management: Threads
- Example: POSIX pthreads