

Concurrency in Distributed Systems

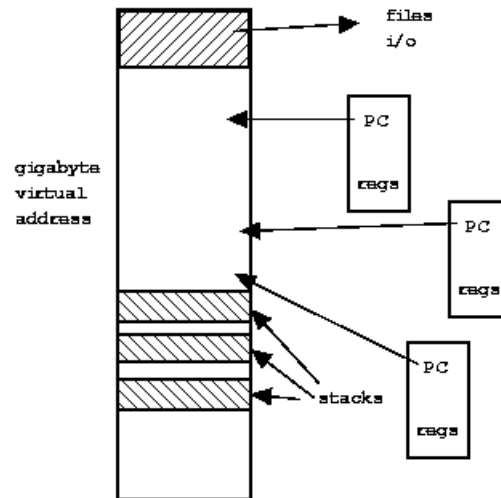
- Part 1: Threads
- Part 2: Concurrency Models
- Part 3: Thread Scheduling

Part 1: Threads and Concurrency

- Traditional process
 - One thread of control through a large, potentially sparse address space
 - Address space may be shared with other processes (shared mem)
 - Collection of systems resources (files, semaphores)
- Thread (light weight process)
 - A flow of control through an address space
 - Each address space can have multiple concurrent control flows
 - Each thread has access to entire address space
 - Potentially parallel execution, minimal state (low overheads)
 - May need synchronization to control access to shared variables

Threads

- Each thread has its own stack, PC, registers
 - Share address space, files,...



Why use Threads?

- Large multiprocessors/multi-core systems need many computing entities (one per CPU or core)
- Switching between processes incurs high overhead
- With threads, an application can avoid per-process overheads
 - Thread creation, deletion, switching cheaper than processes
- Threads have full access to address space (easy sharing)
- Threads can execute in parallel on multiprocessors

Threads Example

```
from time import sleep, perf_counter

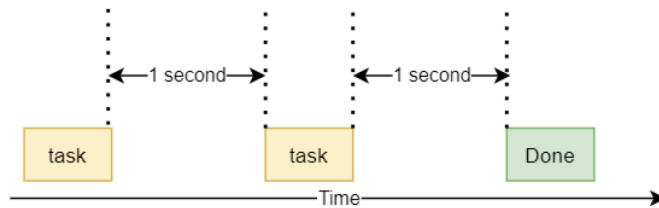
def task():
    print('Starting a task...')
    sleep(1)
    print('done')

start_time = perf_counter()

task()
task()

end_time = perf_counter()
```

Single threaded program



Threads Example

```
from threading import Thread

def task():
    print('Starting a task...')
    sleep(1)
    print('done')

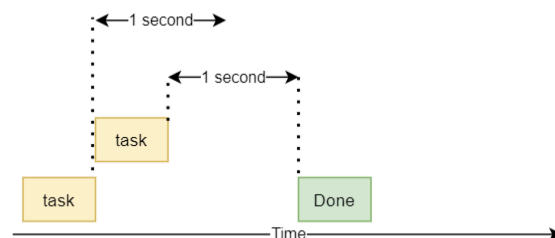
start_time = perf_counter()

# create two new threads
t1 = Thread(target=task)
t2 = Thread(target=task)

# start the threads
t1.start()
t2.start()

# wait for the threads to complete
t1.join()
t2.join()
```

Multi-threaded version



<https://www.pythontutorial.net/advanced-python/python-threading/>

Why Threads?

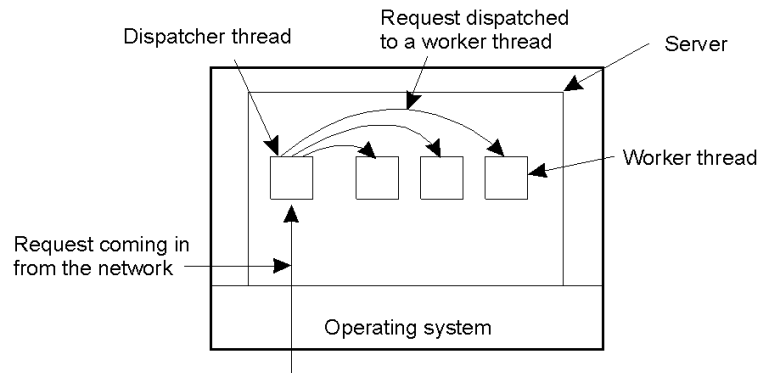
- *Single threaded process*: blocking system calls, no concurrency/parallelism
- *Finite-state machine* [event-based]: non-blocking with concurrency
- *Multi-threaded process*: blocking system calls with parallelism
- Threads retain the idea of sequential processes with blocking system calls, and yet achieve parallelism
- Software engineering perspective
 - Applications are easier to structure as a collection of threads
 - Each thread performs several [mostly independent] tasks

Multi-threaded Clients Example : Web Browsers

- Browsers such as IE are multi-threaded
- Such browsers can display data before entire document is downloaded: performs multiple simultaneous tasks
 - Fetch main HTML page, activate separate threads for other parts
 - Each thread sets up a separate connection with the server
 - Uses blocking calls
 - Each part (gif image) fetched separately and in parallel
 - Advantage: connections can be setup to different sources
 - Ad server, image server, web server...

Multi-threaded Server Example

- Apache web server: pool of pre-spawned worker threads
 - Dispatcher thread waits for requests (“dispatcher-workers” architecture)
 - For each request, choose an idle worker thread
 - Worker thread uses blocking system calls to service web request



Part 2: Concurrency Models

- Concurrency for server-side applications
- All server-side applications involve using a loop to process incoming requests

```
while(1) {  
    wait for incoming request;  
    process incoming request;  
}
```

← called event loop

Sequential Server

- Simplest model: single process, single thread
 - Process incoming requests sequentially

```
while (queue.waitForMessage()) {  
    queue.processNextMessage()  
}
```

- Advantage: very simple
- Disadvantages:
 - Requests queue up while one request is being processed
 - Increases waiting time (queuing delay) and response time

Multi-threaded Server

- Use threads for concurrent processing
- Simple model: **thread per request**
 - For each new request: start new thread, process request, kill thread

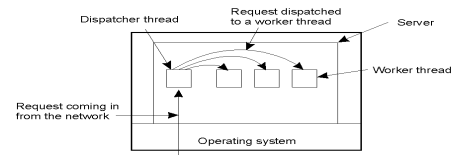
```
while(1) {  
    req = waitForRequest(); // get next request in queue  
                           // wait until one arrives  
    thread = createThread(); // start a new thread  
    thread.process(req); // assign request to thread  
}
```

- Advantage: Newly arriving requests don't need to wait
 - Assigned to a thread for concurrent processing
- Disadvantage: frequent creation and deletion of threads

Server with Thread Pool

- Use **Thread Pool**

- Pre-spawn a pool of threads
- One thread is dispatcher, others are worker threads
- For each incoming request, find an idle worker thread and assign



```
CreateThreadPool (N) ;  
while (1) {  
    req = waitForRequest () ;  
    thread = getIdleThreadfromPool () ;  
    thread.process (req)  
}
```

- Advantage: Avoids thread creation overhead for each request
- Disadvantages:
 - What happens when $>N$ requests arrive at the same time?
 - How to choose the correct pool size N ?

Dynamic Thread Pools

- Optimal size of thread pool depends on request rate
- Online services see dynamic workload
 - Request rate of a web server varies over time
- Dynamic thread pool: vary the number of threads in pool based on workload
 - Start with N threads and monitor number of idle threads
 - If # of idle threads $<$ low threshold, start new threads and add to pool
 - If # of idle threads $>$ high threshold, terminate some threads
- Many modern servers (e.g., apache) use dynamic thread pools to handle variable workloads
 - IT Admin need not worry about choosing optimal N for thread pool

Async Event Loop Model

- Async Event loop servers: single thread but need to process multiple requests
 - Use non-blocking (asynchronous) calls
 - **Asynchronous (aka, event-based) programming**
 - Provide concurrency similar to synchronous multi-threading but with single thread

```
import asyncio

async def count():
    print("One")
    await asyncio.sleep(1)
    print("Two")

async def main():
    await asyncio.gather(count(), count(), count())
```

Async version

```
def count():
    print("One")
    time.sleep(1)
    print("Two")

def main():
    for _ in range(3):
        count()
```

Synchronous version

Event Loop Model

- <https://python.readthedocs.io/en/stable/library/asyncio-eventloop.html>

```
import asyncio

def hello_world(loop):
    print('Hello World')
    loop.stop()

loop = asyncio.get_event_loop()
# Schedule a call to hello_world()
loop.call_soon(hello_world, loop)
# Blocking call interrupted by loop.stop()
loop.run_forever()
loop.close()
```

```
import asyncio

async def hello_world():
    print("Hello World!")

loop = asyncio.get_event_loop()
# Blocking call which returns when the
loop.run_until_complete(hello_world())
loop.close()
```

- async function in python: “coroutine”
- await/async pair `async def foo():` `await: suspend execution of foo`
 `await bar()` `and wait for bar`
- <https://python.plainenglish.io/build-your-own-event-loop-from-scratch-in-python-da77ef1e3c39>
- <https://docs.python.org/3.9/library/asyncio-task.html>

Process Pool Servers

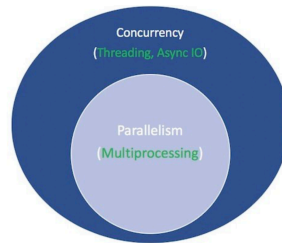
- Multi-process server
 - Use a separate process to handle each request
 - Process Pool: dispatcher process and worker processes
 - Assign each incoming request to an idle process
- Apache web server supports process pools
- Dynamic Process Pools: vary pool size based on workload
- Advantages
 - Worker process crashes only impact the request, not application
 - Address space isolation across workers
- Disadvantages
 - Process switching is more heavy weight than thread switching

Server Architecture

- Sequential
 - Serve one request at a time
 - Can service multiple requests by employing events and asynchronous communication
- Concurrent
 - Server spawns a process or thread to service each request
 - Can also use a pre-spawned pool of threads/processes (apache)
- Thus servers could be
 - Pure-sequential, event-based, thread-based, process-based
- Discussion: which architecture is most efficient?

Parallelism versus Concurrency

- **Concurrency** enables handling of multiple requests
 - Request processing does not block other requests
 - Achieved using threads or async (non-blocking) calls
 - Concurrency can be achieved on single core/processor
- **Parallelism** enable simultaneous processing of requests
 - Does not block other requests; requests processed in parallel
 - Needs multiple threads or multiple processes
 - Threads/processes simultaneously run on multiple cores
 - Async event loops? Will need multiple threads



Part 3: Thread Scheduling

- *Key issues:*
- Cost of thread management
 - More efficient in user space
- Ease of scheduling
- Flexibility: many parallel programming models and schedulers
- Process blocking – a potential problem

User-level Threads

- Threads managed by a threads library
 - Kernel is unaware of presence of threads
- Advantages:
 - No kernel modifications needed to support threads
 - Efficient: creation/deletion/switches don't need system calls
 - Flexibility in scheduling: library can use different scheduling algorithms, can be application dependent
- Disadvantages
 - Need to avoid blocking system calls [all threads block]
 - Threads compete for one another
 - Does not take advantage of multiprocessors [no real parallelism]

User-level threads

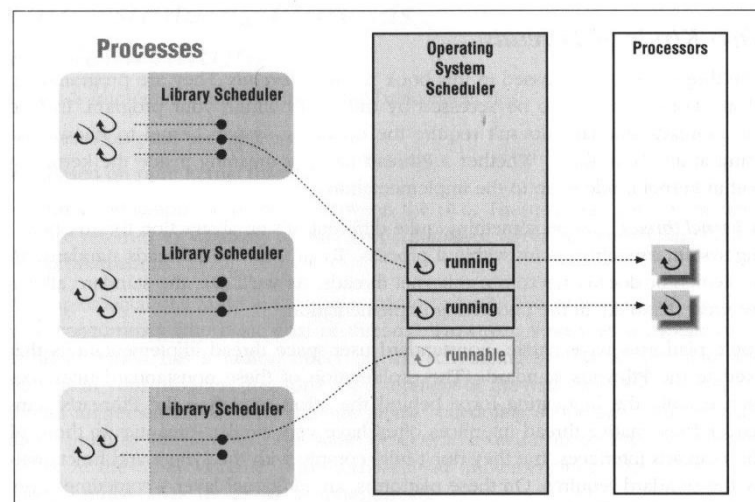


Figure 6-1: User-space thread implementations

Kernel-level threads

- Kernel aware of the presence of threads
 - Better scheduling decisions, more expensive
 - Better for multiprocessors, more overheads for uniprocessors

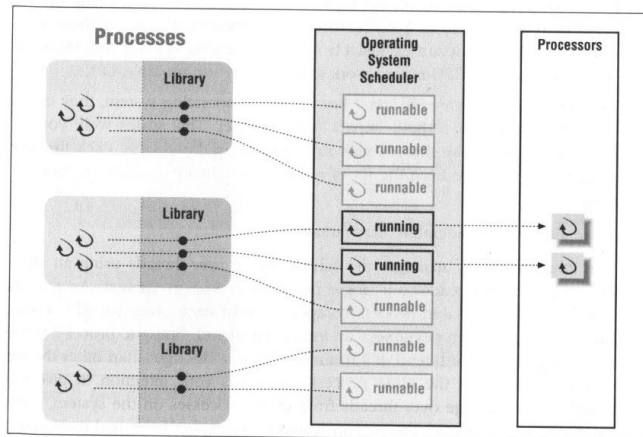
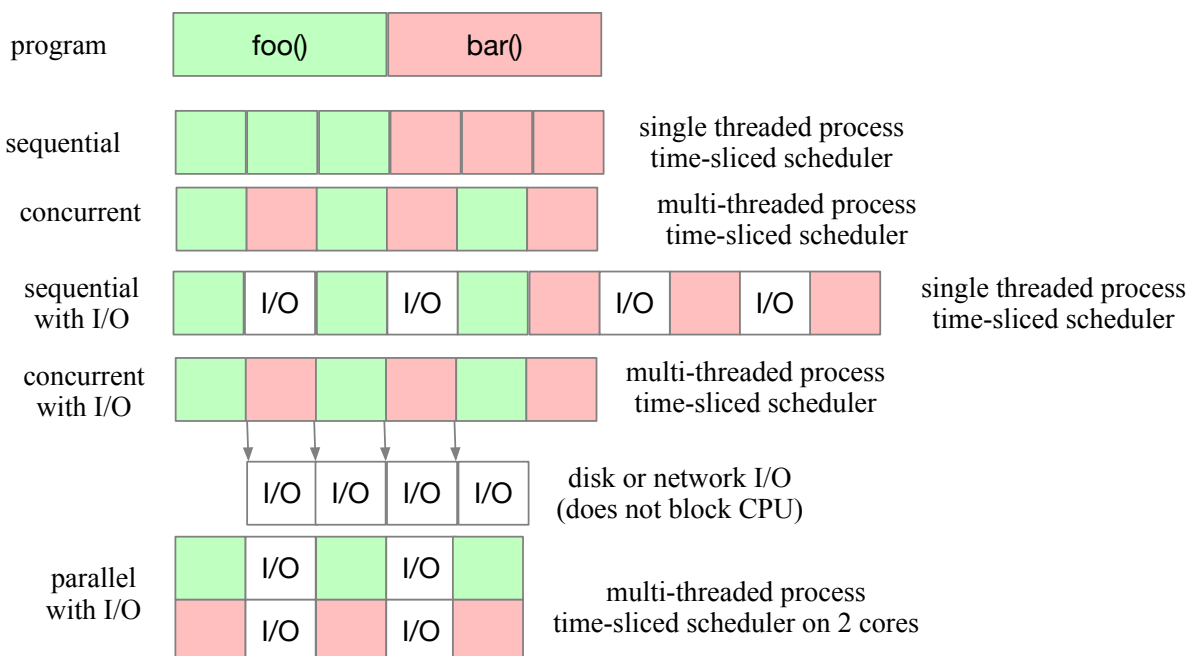


Figure 6-2: Kernel thread-based implementations

Thread Scheduling Example

- CPU scheduler uses round-robin time slices



Scheduler Activation

- User-level threads: scheduling both at user and kernel levels
 - user thread system call: process blocks
 - kernel may context switch thread during important tasks
- Need mechanism for passing information back and forth
- Scheduler activation: OS mechanism for user level threads
 - Notifies user-level library of kernel events
 - Provides data structures for saving thread context
- Kernel makes up-calls : CPU available, I/O is done etc.
- Library informs kernel: create/delete threads
 - N:M mapping: n user-level threads onto M kernel entities
- Performance of user-level threads with behavior of kernel threads

Light-weight Processes

- Several LWPs per heavy-weight process
- User-level threads package
 - Create/destroy threads and synchronization primitives
- Multithreaded applications – create multiple threads, assign threads to LWPs (one-one, many-one, many-many)
- Each LWP, when scheduled, searches for a runnable thread [*two-level scheduling*]
 - Shared thread table: no kernel support needed
- When a LWP thread block on system call, switch to kernel mode and OS context switches to another LWP

LWP Example

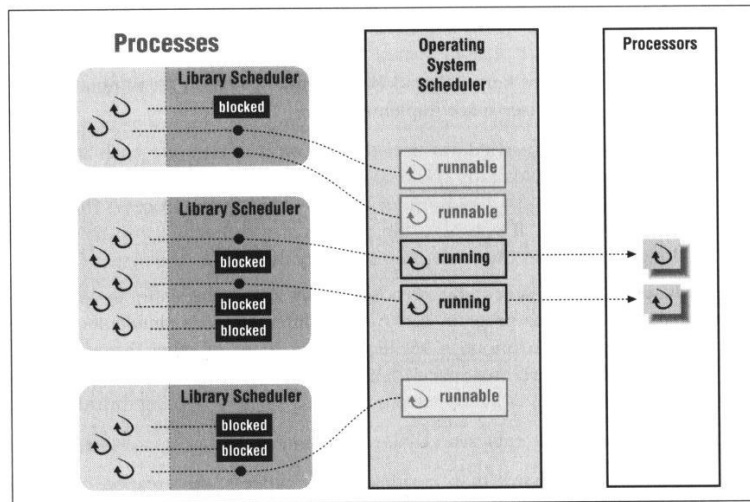


Figure 6-3: Two-level scheduler implementations

Process Scheduling

- Priority queues: multiples queues, each with a different priority
 - Use strict priority scheduling
 - Example: page swapper, kernel tasks, real-time tasks, user tasks
- Multi-level feedback queue
 - Multiple queues with priority
 - Processes dynamically move from one queue to another
 - Depending on priority/CPU characteristics
 - Gives higher priority to I/O bound or interactive tasks
 - Lower priority to CPU bound tasks
 - Round robin at each level

