Concurrency: Running Together

ECE 650
Methods & Tools for Software Engineering (MTSE)
Fall 2023

Presented by
Dr. Albert Wasef

Used by permission from Prof. Arie Gurfinkel

UNIVERSITY OF
WATERLOO

MULTIPROGRAMMING

Multiprogramming

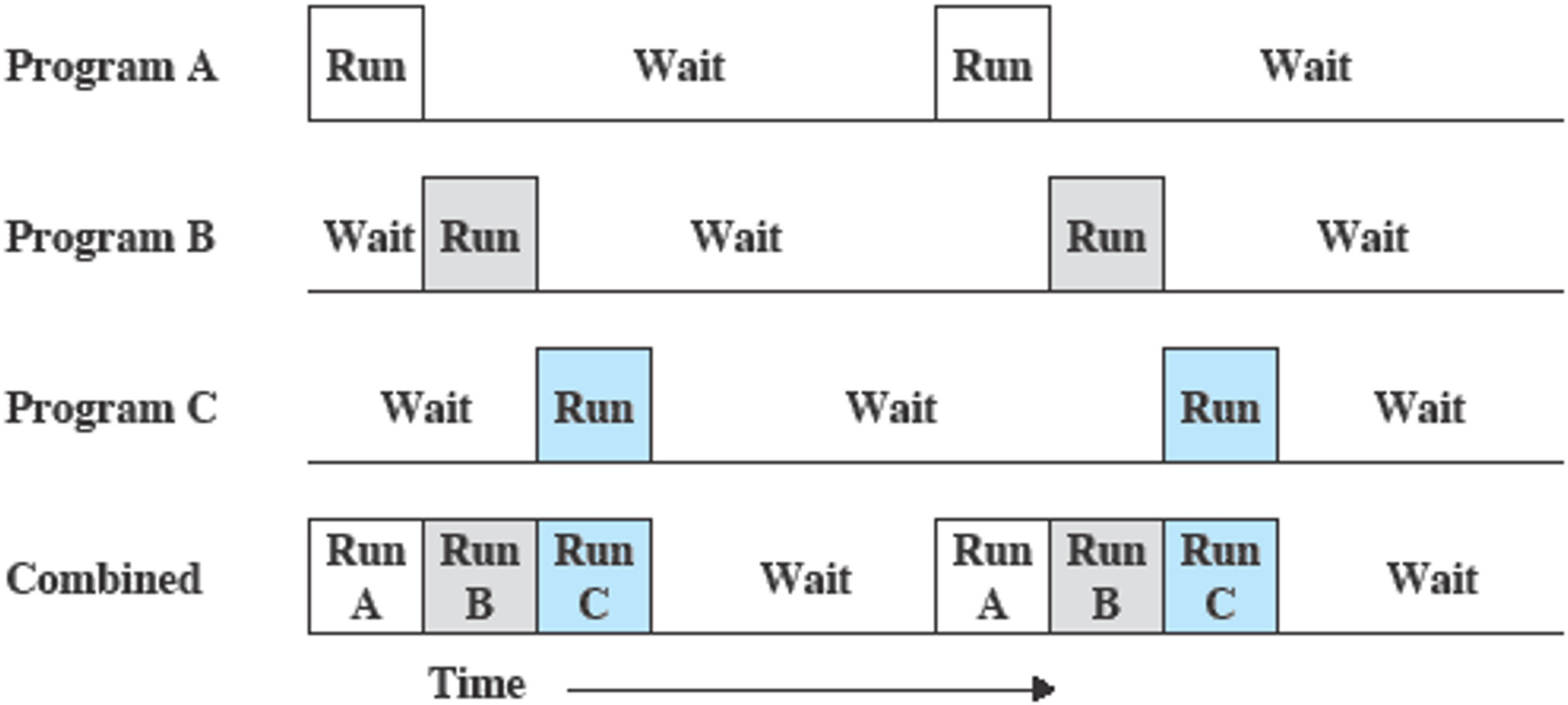
Concurrent execution of multiple tasks (e.g., processes)

- Each task runs as if it was the only task running on the CPU

Benefits:

- When one task needs to wait for I/O, the processor can switch to another task
- (why is this potentially a huge benefit?)

Multiprogramming



(c) Multiprogramming with three programs

Multiprogramming: Example

Web-application

- Python web-server that handles web requests and generates jobs
- Multi-process job-server that processes jobs in parallel

Web-server and Job-server communicate via the file system

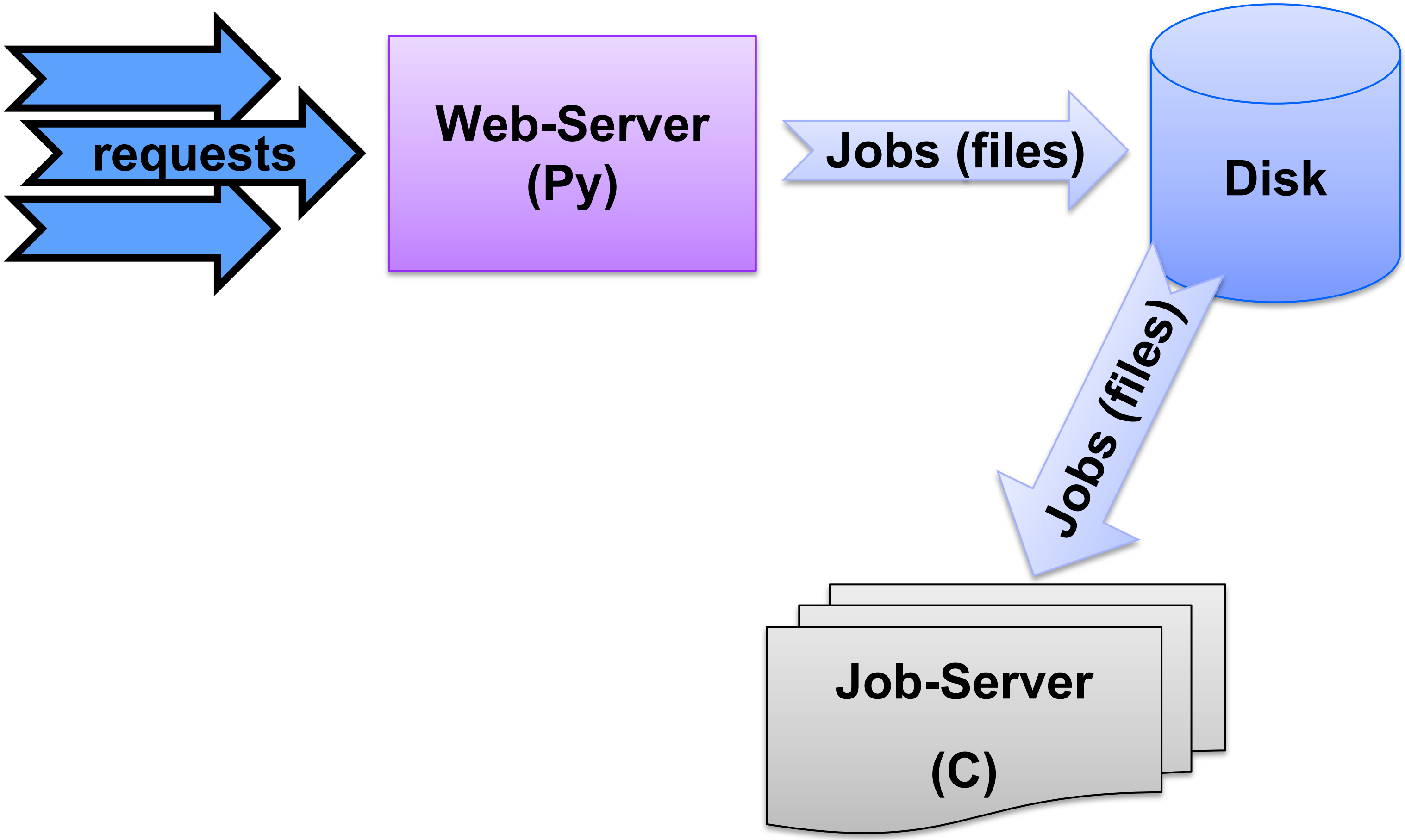
Workers in Job-server compete to acquire jobs to process

Potential problem: Race Condition

- Two workers attempt to acquire the same job to process
- Only one will succeed
  - better than both completing the same job

https://git.uwaterloo.ca/ece650-f23/threads/-/tree/master/webapp

Web-Application: Architecture



Race Condition

A situation where concurrent operations access data in a way that the outcome depends on the order (the timing) in which operations execute.

- **Not** necessarily a **bug**!
- Often is the main source of bugs in concurrent systems
- Programmers assume that order of execution does not influence the result, and/or, implicitly assume that only certain order of operations is possible

In our web-app example, workers are **racing** to rename/lock a job file

- only one succeeds, so the race is not causing a bug
- but it does create unexpected behaviour (disappearing file)

MULTITHREADING

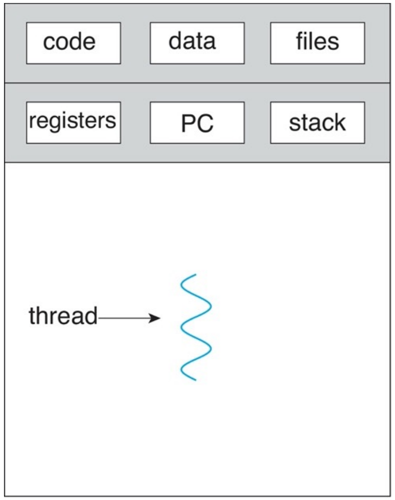
Traditional UNIX Process

Process is OS abstraction of what is needed to run single program

- Often called "heavyweight process"

Processes have two parts

- Sequential program execution stream (active part)
  - Code executed as sequential stream of execution (i.e., thread)
  - Includes state of CPU registers
- Protected resources (passive part)
  - Main memory state (contents of Address Space)
  - I/O state (i.e. file descriptors)



single-threaded process

Modern Process with Threads

Thread: sequential execution stream within process
(sometimes called "lightweight process")

- Process still contains single address space
- No protection between threads

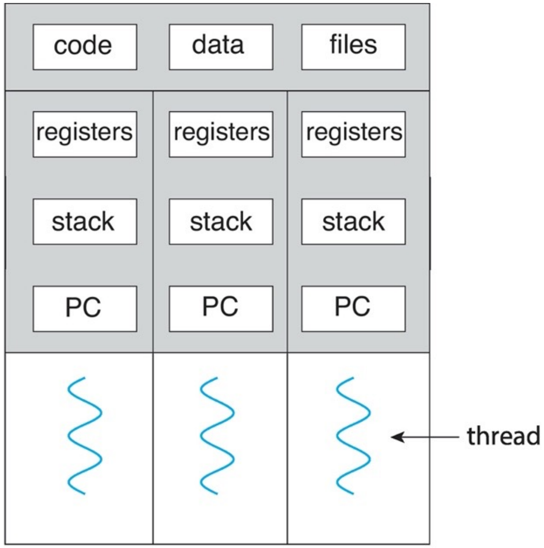
Multithreading: single program made up of different concurrent activities (sometimes called multitasking)

Some states are shared by all threads

- Content of memory (global variables, heap)
- I/O state (file descriptors, network connections, etc.)

Some states "private" to each thread

- CPU registers (including PC) and stack



multithreaded process

Threads Motivation

OS's need to handle multiple things at once (MTAO)

- Processes, interrupts, background system maintenance

Servers need to handle MTAO

- Multiple connections handled simultaneously

Parallel programs need to handle MTAO

- To achieve better performance

Programs with user interfaces often need to handle MTAO

- To achieve user responsiveness while doing computation

Network and disk programs need to handle MTAO

- To hide network/disk latency

Multithreading: Process versus Thread

Process provides an execution context for the program

- **unit of ownership**
- Memory, I/O resources, console, etc.
- Process pretends like it is a single entity controlling the execution environment
- Inter Process Communication (IPC) is “like” communicating between individual machines (but connected with super-fast network)

Thread represent a single execution unit (i.e., CPU)

- unit of scheduling
- ancient time: a process has one thread running on one physical CPU
- old time: a process has many threads sharing one physical CPU
- today: a process has many threads sharing many physical CPUs (multicore)
- all threads of a process share the same memory space!

Threads: Programmer's Perspective

A thread is a function that is ran concurrently with other functions

- It is like `fork()` followed by a call to a child process function
- **Except**: no new process is created. The new thread can access **all** the data of the **current** process

```c
void * foo(void*) {...}
void * bar(void*) {...}

int main(void) {
  pthread_t t1, t2;
  void *data;
  ...
  pthread_create(&t1, NULL, foo, data);
  pthread_create(&t2, NULL, bar, data);

  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
}
```

Threads: Programmer's Perspective

A thread is a function that is ran concurrently with other functions

- It is like fork() followed by a call to a child process function
- Except: no new process is created. The new thread can access all the data of the **current** process

```
void * foo(void*) {...}
void * bar(void*) {...}

int main(void) {
  pthread_t t1, t2;
  void *data;
  ...
  pthread_create(&t1, NULL, foo, data);
  pthread_create(&t2, NULL, bar, data);

  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
}
```

Code that will execute concurrently

Start of concurrent execution

Main thread waits for others to finish

Multithreading Example: Compute CRC

CRC – Cyclic Redundancy Check is an error detecting code used to identify error in data

- `CRC(data) = crc_code`,
  - where `crc_code` is a ”small” number summarizing `data`
- CRC computation is expensive, but easy to parallelize

Parallel Computation of CRC

- Divide data into chunks: `data1, data2, data3, …`
- Compute CRC for each chunk
  - `crc1 = CRC(data1), crc2 = CRC(data2), crc3 = CRC(data3), …`
- Combine CRC of chunks into CRC of the data
  - `crc_code = crc1 ++ crc2 ++ crc3 ++ …`
- CRC of each chunk is computed in parallel (using threads)

https://git.uwaterloo.ca/ece650-f23/threads/-/tree/master/checksum

References

Slides & Demo credit:

- Carlos Moreno (cmoreno@uwaterloo.ca)
- Reza Babaee
- Prof. Seyed M. Zahedi (ECE350 UWaterloo)