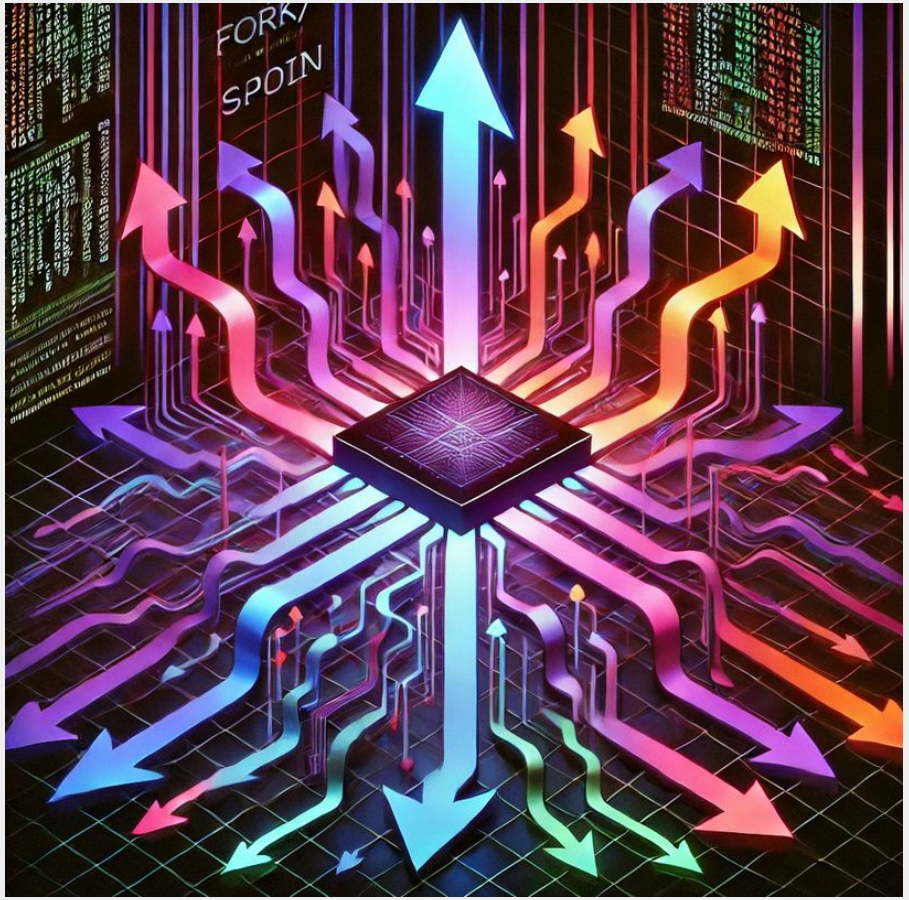


UNSW SYDNEY

COMP1521 26T3

Processes, Pipes, & Pthreads Aka Concurrency, Parallelism and Threads

2026



Announcements

- **Week 10 lab Practice Exam**
  - In person lab classes
    - Please attend one even if you are usually online
  - Questions are not released you need to attend to see them
  - Not worth marks
  - Regular lab questions for marks to do too
- **Assignment 2**: Get started ASAP if you have not already.
  - Start now while
    - Help sessions and forums are not super busy!
    - You still have a chance to get help in tut/labs!
  - Look at style rubric to see what we are looking for

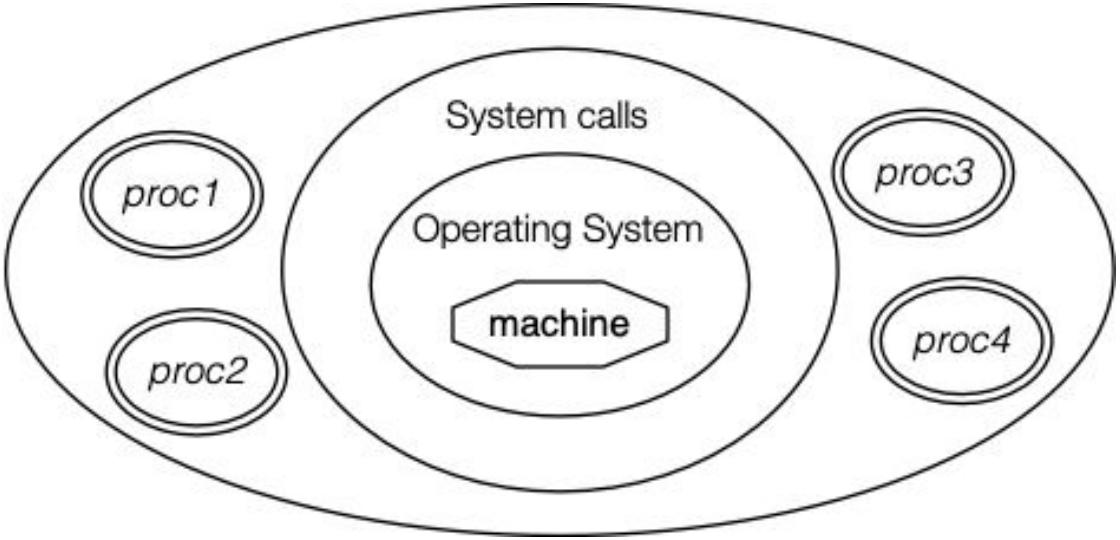
Recap Exercise

Today's Lecture

- Recap: Processes
- Pipes
- Concurrency
- Threads
  - Mutexes
  - Atomics

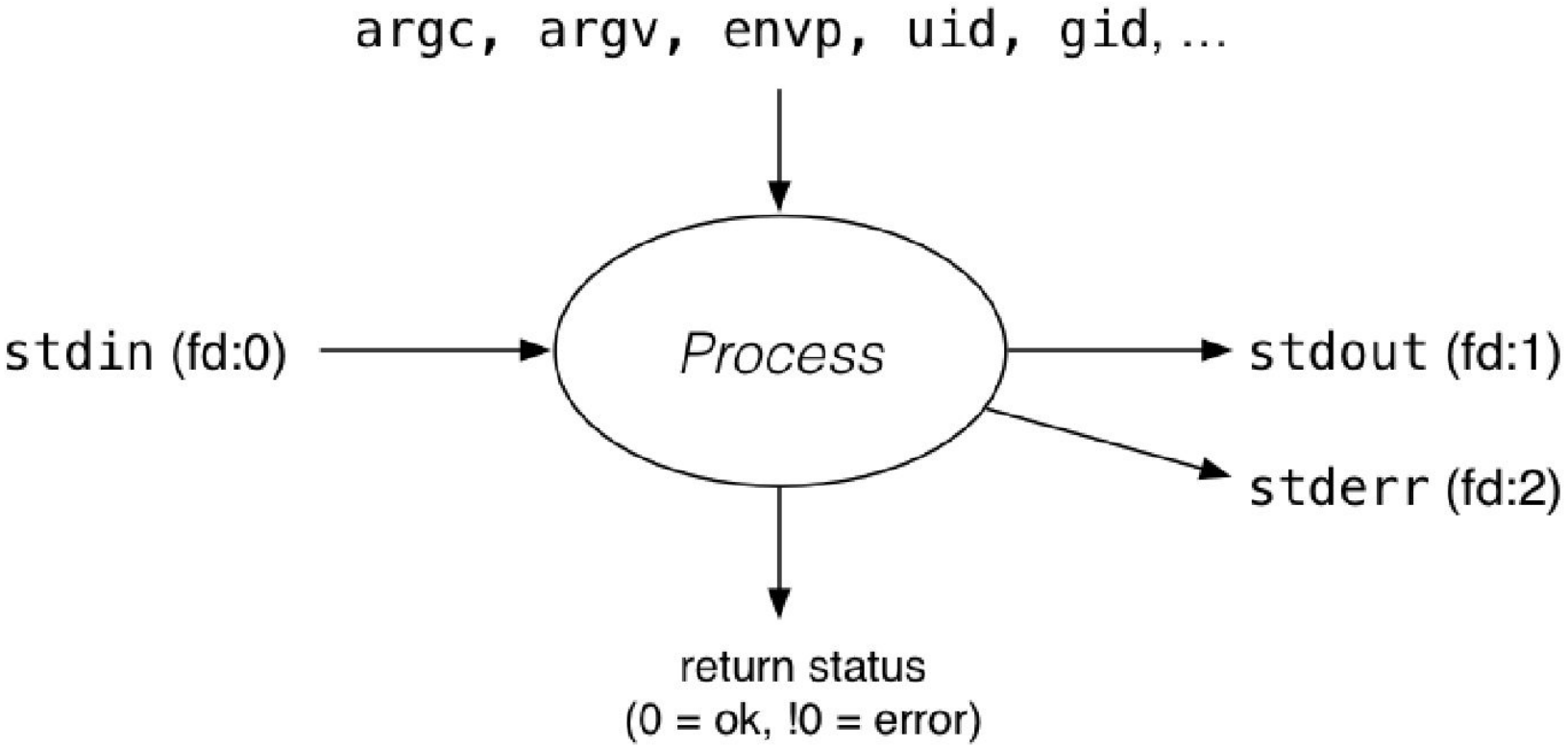


Recap: A computer process



- A process is a program running in an environment
- The operating system manages starting, stopping processes

Recap: Env. for Unix/Linux Processes



Recap: Processes

- A process is an instance of an executing program.

- Each process has an execution state, defined by…
  - Current values of **CPU registers**
  - Current contents of its **memory**
  - Information about **open files** (and other results of system calls)

Recap: syscalls to get info about a process

- **`pid_t getpid()`**
  - requires `#include <sys/types.h>`
  - returns the process ID of the current process

- **`pid_t getppid()`**
  - requires `#include <sys/types.h>`
  - returns the parent process ID of the current process

- For more details: man 2 getpid
- Not used in this course: `getpgid()` … get process group ID

Minimal example for getpid() and getppid():

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main(void){
    printf("My PID is (%d)\n", getpid());
    printf("My parent's PID is (%d)\n", getppid());
    return 0;
}
```

Recap: Environment variables

- Unix-like shells have simple syntax to set environment variables
  - You can type `env` on the command line to see them all
  - Common to set environment in startup files (e.g .profile)
  - Then passed to any programs they run
  - Almost all program pass the environment variables they are given to any programs they run
    - They perhaps add/edit the value of specific environment variables

setenv() - set an environment variable

```
#include <stdlib.h>
int setenv(const char *name, const char *value, int overwrite);
```

- adds name=value to environment variable array
- if name in array, value changed if overwrite is non-zero

Returns 0 if success, or -1 if error (error stored in *errno*)

getenv() - get an environment variable

```c
#include <stdlib.h>
char *getenv(const char *name);
```

- Reads value from environment variable array by name
- if name not in array, returns NULL

Example: to get home_directory (aka ~)

```c
printf("%s", getenv("HOME"));
```

Demo: get_env.c, get_status.c

Multi-Tasking

On a typical modern operating system…

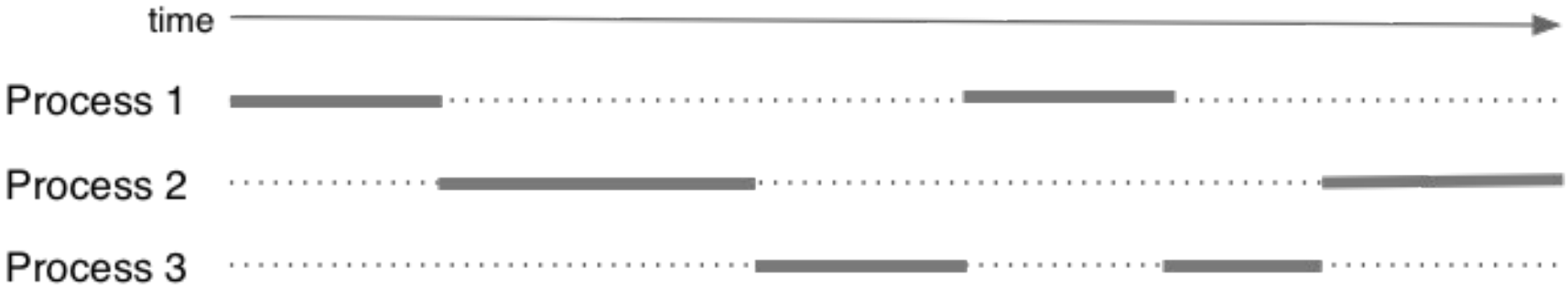
- multiple processes are active “simultaneously” (multi-tasking)
- operating systems provides a virtual machine to each process:
- each process executes as if it is the only process running
- e.g. each process has its own **address space** (N bytes, addressed 0..N-1)

Multi-Tasking (cont.)

When there are multiple processes running on the machine,

- a process uses the CPU, until it is **preempted** or **exits**;
- then, another process uses the CPU, until it too is preempted.
- eventually, the first process will get another run on the CPU.

Multi-Tasking (cont.) (cont.)



Overall impression: three programs running simultaneously.

(In practice, these time divisions are imperceptibly small!)

Recap: Making Processes

Scary, unsafe way **system()**

Old-fashioned way **fork()** then **exec()**

- **fork()** duplicates the current process (parent+child)
- **execve()** “overwrites” the current process (run by child)

New, standard way **posix_spawn()**

posix_spawn() — Run a new process

```
#include <spawn.h>
int posix_spawn(
    pid_t *pid, const char *path,
    const posix_spawn_file_actions_t *file_actions,
    const posix_spawnattr_t *attrp,
    char *const argv[], char *const envp[]);
```

- **pid**: returns process id of new program
- **path**: path to the program to run
- file_actions: specifies file actions to be performed before running program - can be used to redirect stdin, stdout to file or pipe

posix_spawn() — Run a new process

```
#include <spawn.h>
int posix_spawn(
    pid_t *pid, const char *path,
    const posix_spawn_file_actions_t *file_actions,
    const posix_spawnattr_t *attrp,
    char *const argv[], char *const envp[]);
```

- attrp: specifies attributes for new process (not covered in COMP1521)
- **argv**: arguments to pass to new program
- **envp**: environment to pass to new program

can also use **posix_spawnp** which searches PATH

Example: posix_spawn() to run /bin/date

```c
pid_t pid;
extern char **environ;
char *date_argv[] = {"/bin/date", "--utc", NULL};
// spawn "/bin/date" as a separate process
int ret = posix_spawn(&pid, "/bin/date", NULL, NULL, date_argv, environ);
if (ret != 0) {
    errno = ret; //posix_spawn returns error code, does not set errno
    perror("spawn"); exit(1);
}
// wait for spawned processes to finish
int exit_status;
if (waitpid(pid, &exit_status, 0) == -1) {
    perror("waitpid"); exit(1);
}
printf("/bin/date exit status was %d\n", exit_status);
```

Demo: spawn.c

Example: posix_spawn() versus system()

Running **ls -ld** via **posix_spawn()**

```
char *ls_argv[2] = {"/bin/ls", "-ld", NULL};
pid_t pid; int ret;
extern char **environ;
if((ret = posix_spawn(&pid, "/bin/ls", NULL, NULL, ls_argv, environ)) != 0) {
    errno = ret; perror("spawn"); exit(1);
}
int exit_status;
if (waitpid(pid, &exit_status, 0) == -1) {
    perror("waitpid"); exit(1);
}
```

Inter-process Communication

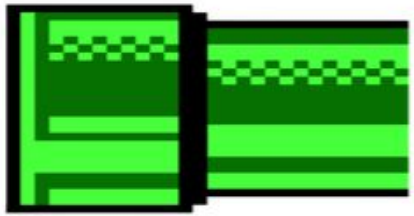
Inter-Process Communication(IPC)

- Processes have their own address spaces
- IPC allows processes to share data with each other.
- **Pipes** are just one form of IPC.
- There are others you may learn about in other courses.

pipe() — stream bytes between processes

Send output of one process as input to another

Process A
WRITES
to the
pipe



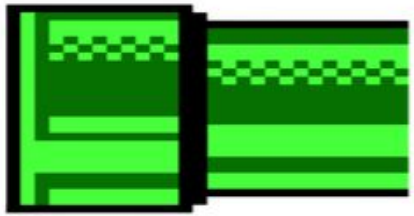
Process B
READS
from the
pipe

pipe() — stream bytes between processes

Demo: on the command line:

```
seq 1 10 | wc
```

Process A WRITES to the pipe



Process B READS from the pipe

pipe() — stream bytes between processes

```c
#include <unistd.h>
int pipe(int pipefd[2]);
```

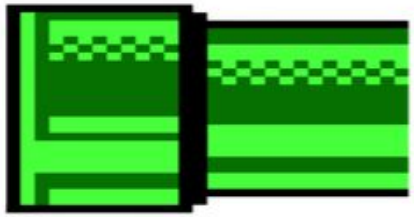
Pipes: unidirectional byte streams provided by operating system

- **pipefd[0]**: set to file descriptor of read end of pipe
- **pipefd[1]**: set to file descriptor of write end of pipe
- bytes written to **pipefd[1]** will be read from **pipefd[0]**

Child processes (by default) inherit file descriptors including pipes

pipe() — stream bytes between processes

Process A
WRITES
to
pipefd[1]



Process B
READS
from
pipefd[0]

Closing pipes

Parent can send/receive bytes (not both) to child via pipe

- parent and child should both close unused pipe file descriptors
- e.g if bytes being written (sent) parent to child
    - parent should close read end **pipefd[0]**
    - child should close write end **pipefd[1]**

Pipe file descriptors can be used with stdio via **fdopen()**

popen() — convenient way to set up pipe

```
#include <stdio.h>
FILE *popen(const char *command, const char *type);
int pclose(FILE *stream);
```

- runs command via /bin/sh
- if type is “w” pipe to stdin of command created
- if type is “r” pipe from stdout of command created
- FILE * stream returned - get then use fgetc/fputc etc
- NULL returned if error
- close stream with pclose (not fclose)
- pclose waits for command and returns exit status

popen() — a bit unsafe

```c
#include <stdio.h>
FILE *popen(const char *command, const char *type);
int pclose(FILE *stream);
```

- convenient but brittle
- vulnerable to command injection (same as system())
- try to avoid use except in debugging and throw-away programs

Example: process output with popen()

```c
// popen passes string to a shell for evaluation
// brittle and highly-vulnerable to security exploits
// popen is suitable for quick debugging and throw-away programs only
FILE *p = popen("/bin/date --utc", "r");
if (p == NULL) {
   perror(""); return 1;
}
char line[256];
if (fgets(line, sizeof line, p) == NULL) {
   fprintf(stderr, "no output from date\n"); return 1;
}
printf("output captured from /bin/date was: '%s'\n", line);
pclose(p); // returns command exit status
```

Demo: read_popen.c

Example: input to a process with popen()

```c
int main(void) {
    // popen passes command to a shell for evaluation
    // brittle and highly-vulnerable to security exploits
    //
    // tr a-z A-Z - passes stdin to stdout converting lower case to upper
case
    FILE *p = popen("tr a-z A-Z", "w");
    if (p == NULL) {
        perror("");
        return 1;
    }
    fprintf(p, "hello, i am a COMP1521 aficionado\n");
    pclose(p); // returns command exit status
    return 0;
}
```

Demo: write_popen.c

posix_spawn and pipes (advanced topic)

```
int posix_spawn_file_actions_destroy(
   posix_spawn_file_actions_t *file_actions);
int posix_spawn_file_actions_init(
   posix_spawn_file_actions_t *file_actions);
int posix_spawn_file_actions_addclose(
   posix_spawn_file_actions_t *file_actions, int fildes);
int posix_spawn_file_actions_adddup2(
   posix_spawn_file_actions_t *file_actions, int fildes, int newfildes);
```

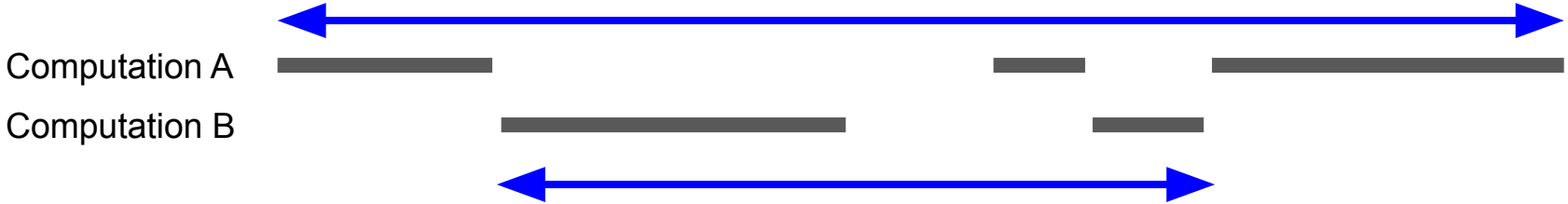
- functions to combine file ops with posix_spawn process creation
- awkward to understand and use — but robust

Example: capturing output from a process: spawn_read_pipe.c
Example: sending input to a process: spawn_write_pipe.c

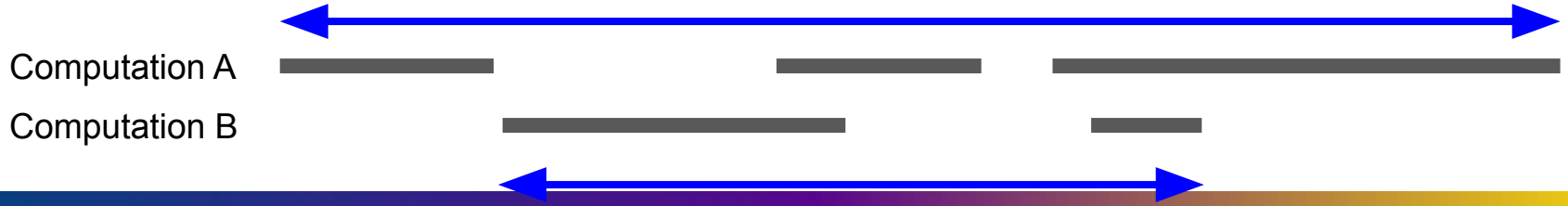
Concurrency

Concurrency & Parallelism

Concurrency: Multiple computations with *overlapping* time periods. Does not *have* to be simultaneous.



Parallelism: Multiple computations executing *simultaneously*.



Question

Have we already seen concurrency in this course?

What about parallelism?

Very High-Speed Computing Systems

MICHAEL J. FLYNN, MEMBER, IEEE

Abstract—Very high-speed computers may be classified as follows:

1) Single Instruction Stream–Single Data Stream (SISD)
2) Single Instruction Stream–Multiple Data Stream (SIMD)
3) Multiple Instruction Stream–Single Data Stream (MISD)
4) Multiple Instruction Stream–Multiple Data Stream (MIMD).

"Stream," as used here, refers to the sequence of data or instructions as seen by the machine during the execution of a program.

The constituents of a system: storage, execution, and instruction handling (branching) are discussed with regard to recent developments and/or systems limitations. The constituents are discussed in terms of concurrent SISD systems (CDC 6600 series and, in particular, IBM Model 90 series), since multiple stream organizations usually do not require any more elaborate components.

Representative organizations are selected from each class and the arrangement of the constituents is shown.

Manuscript received June 30, 1966; revised August 16, 1966. This work was performed under the auspices of the U. S. Atomic Energy Commission.

The author is with Northwestern University, Evanston, Ill., and Argonne National Laboratory, Argonne, Ill.

INTRODUCTION

MANY SIGNIFICANT scientific problems require the use of prodigious amounts of computing time. In order to handle these problems adequately, the large-scale scientific computer has been developed. This computer addresses itself to a class of problems characterized by having a high ratio of computing requirement to input/output requirements (a partially de facto situation

Flynn's Taxonomy for Classifying Parallelism

SISD: Single Instruction, Single Data ("no parallelism")

- e.g. mipsy

SIMD: Single Instruction, Multiple Data ("vector processing")

- Multiple cores of a CPU executing (parts of) same instruction
- e.g. GPUs (graphics rendering and training and running neural networks e.g. LLMs)

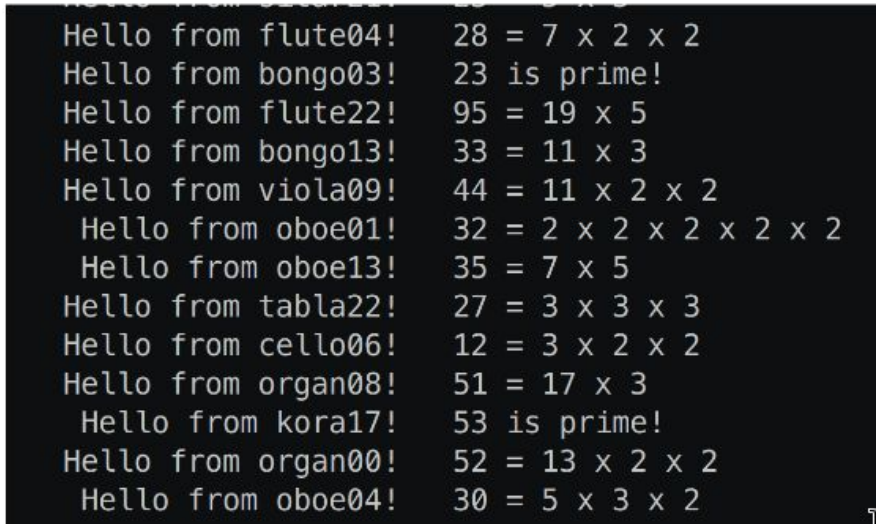
MISD: Multiple Instruction, Single Data

- e.g., fault tolerance in space shuttles (task replication)

MIMD: Multiple Instruction, Multiple Data ("multiprocessing")

Parallel computing

- Distributing computation across multiple computers
  - One popular framework is MapReduce
  - Necessary for *very big* computations and *very large* sets of data
  - Can be difficult to deal with synchronisation and failure of machines
  - Out of scope for COMP1521



Concurrency with processes

- Create multiple processes, and split the job across them
- Each process
  - runs concurrently
  - has its **own address space** (giving **isolation**)
    - Helps with security and robustness
- Processes can be distributed across cores, giving parallelism
- But this strategy is expensive!
  - Creation/teardown expensive
  - Switching expensive
  - Lots of state per process ⇒ costs memory

Threads

Threads: concurrency *within* a process

- Threads allows us to create concurrency ***within*** a process
- Threads within a process ***share*** the **address space**:
  - threads share code
  - threads share global variables
  - threads share the heap (`malloc`)
  - cheap communication!
- Some other process state is shared
  - environment variables, file descriptors, current working directory, …

Threads: concurrency *within* a process

- Threads allows us to create concurrency ***within*** a process
- Each thread has a separate execution state
  - Separate **registers**, separate **program counter**
- Each thread has a **separate stack**
  - but a thread can still read/write to another thread's stack
- Each thread gets its own copy of errno!

Using POSIX Threads (pthreads)

- POSIX Threads is a widely-supported threading model
- Provides an API/model for managing threads (and synchronisation)

```
#include <pthread.h>
```

- Sometimes need **`-pthread`** when compiling
- C11 and later also provides a model/API similar to pthreads
    - Has some small differences with pthreads, and generally less-supported and less used (for now…)

Creating threads with `pthread_create`

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

- Starts a new thread running `start_routine(arg)`
- An ID for the thread is stored in `thread`
- Thread has attributes specified in `attr` (NULL if you don't want special attributes)
- Returns 0 if OK, otherwise an error number (**does not set errno!**)
- Analogous to ***posix_spawn***.

Examples

- one_thread_infinite_loops.c
- one_thread.c

Waiting for threads with `pthread_join`

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

- Waits for `thread` to terminate, if it hasn't already terminated
- Return/exit value of thread placed in `*retval`
- Analogous to `waitpid`

- When **main** returns, *all* threads terminate

More examples

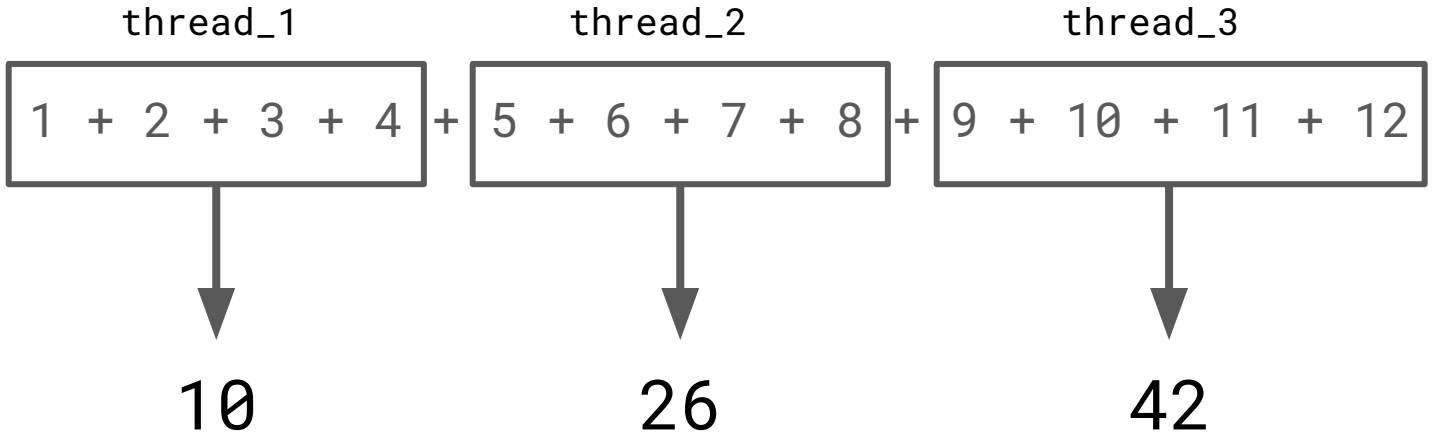
- two_threads_broken.c
- two_threads.c
- nthreads.c
- threads_return.c

Something Useful with Threads!

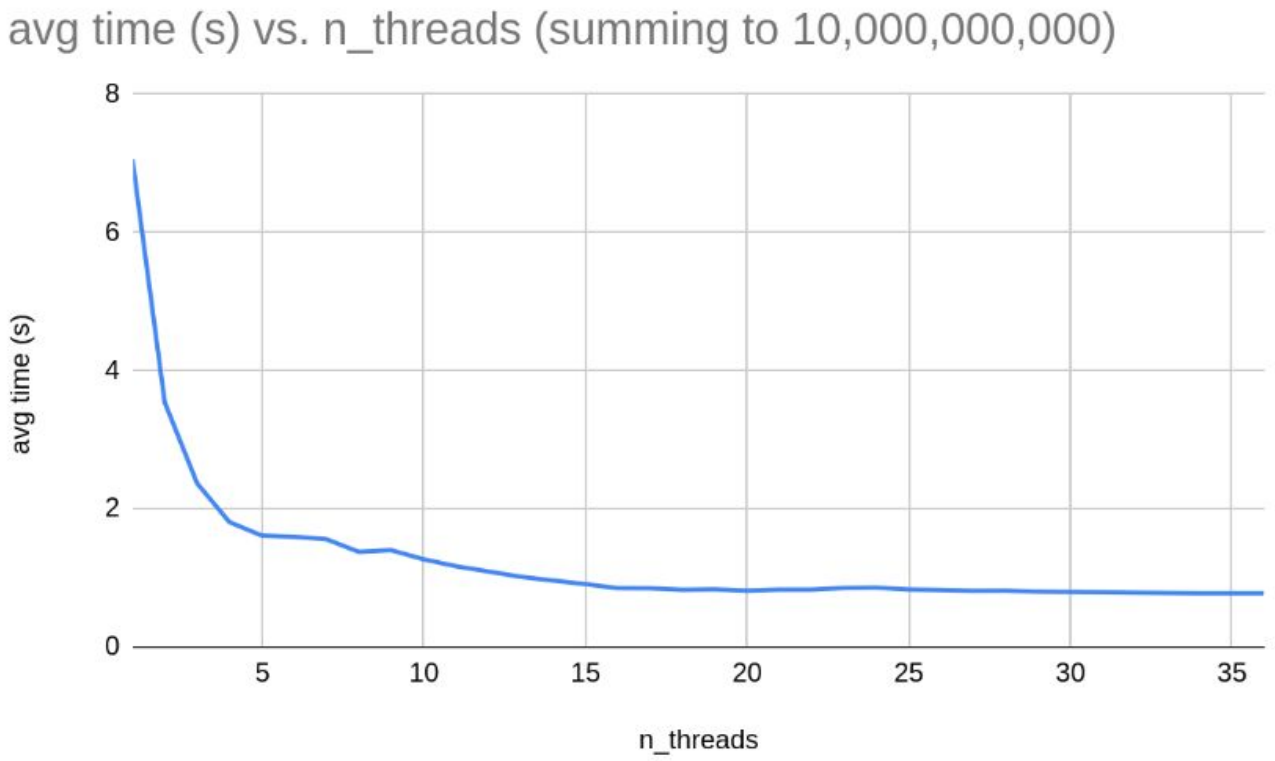
naive_sum.c

thread_sum.c

Example thread_sum



A graph of the performance of thread_sum.c



Some other concurrency benefits

- One thread can wait for I/O (block) while others make progress or wait for separate I/O
- Useful for user interface programming

Data Lifetime Issues

- When sharing data with a thread we pass in **addresses of data**
  - What if by the time the thread reads the data, that data no longer exists?
- So far we have put data in local variables in main
  - Main outlives all of the created threads
- What if we create threads from functions other than main?

- Demo: thread_data_broken.c
- Demo: thread_data_malloc.c

Data Races, Deadlock and Disasters

Unsafe Access to Global Variables

Demo: bank_account_broken.c

Incrementing a global variable is **NOT** an atomic operation

```c
int bank_account;

void *thread(void *a) {
    // ...
    bank_account++;
    // ...
```

```
la   $t0, bank_account
lw   $t1, ($t0)
addi $t1, $t1, 1
sw   $t1, ($t0)
.data
bank_account:  .word 0
```

Global Variables and Race Condition

If bank_account = 42 and two threads execute concurrently

```
la      $t0, bank_account
# {| bank_account = 42 |}
lw      $t1, ($t0)
# {| $t1 = 42 |}
addi    $t1, $t1, 1
# {| $t1 = 43 |}
sw      $t1, ($t0)
# {| bank_account = 43 |}
```

```
la      $t0, bank_account
# {| bank_account = 42 |}
lw      $t1, ($t0)
# {| $t1 = 42 |}
addi    $t1, $t1, 1
# {| $t1 = 43 |}
sw      $t1, ($t0)
# {| bank_account = 43 |}
```

Oops! We lost an increment.
Threads share global variables!

Global Variables and Race Condition

If bank_account = 100 and two threads execute concurrently

```
la      $t0, bank_account
# {| bank_account = 100 |}
lw      $t1, ($t0)
# {| $t1 = 100 |}
addi    $t1, $t1, 100
# {| $t1 = 200 |}
sw      $t1, ($t0)
# {| bank_account = ...? |}
```

```
la      $t0, bank_account
# {| bank_account = 100 |}
lw      $t1, ($t0)
# {| $t1 = 100 |}
addi    $t1, $t1, -50
# {| $t1 = 50 |}
sw      $t1, ($t0)
# {| bank_account = 50 or 200 |}
```

- This is a critical section.
- We don’t want two threads in the critical section
  - We must establish mutual exclusion.

A solution: mutexes

- We need a way of guaranteeing *mutual exclusion* for certain shared resources (such as ***bank_account***)
- We associate each of those resources with a ***mutex***
- Only one thread can hold a mutex, any other threads which attempt to lock the mutex must wait until the mutex is unlocked
- So only one thread will be executing the section between the mutex lock and the mutex unlock

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

bank_account_mutex.c

```
int bank_account=0;
pthread_mutex_t bank_account_lock=PTHREAD_MUTEX_INITIALIZER;

void *add_100000(void*argument){
    for (int i = 0; i < 1000000; i++) {
        pthread_mutex_lock(&bank_account_lock);
        // only one thread can execute this
        // section of code at any time
        bank_account = bank_account + 1;
        pthread_mutex_unlock(&bank_account_lock);
    }
```

Mutex the world!

- Mutexes solve all our data race problems!
- So… just put a mutex around everything!
- This works… but then we lose the advantages of parallelism
- Mutexes also have overhead
- Python 🐍 does this
  - Global Interpreter Lock (GIL) (although can be disabled now…)
- Linux used to do this (they removed the ‘Big Kernel Lock’ in 2011)

Code Demo: Deadlock

bank_account_deadlock.c

Deadlocks

- No thread can make progress!
- The system is deadlocked

THREAD 1

1. acquire lock_A
2. acquire lock_B ❌ **BLOCKED!**
3. do_somthing(A, B)
4. release lock_B
5. release lock_A

THREAD 2

1. acquire lock_B
2. acquire lock_A ❌ **BLOCKED!**
3. do_somthing(A, B)
4. release lock_A
5. release lock_B

`lock_A` 🔒

`lock_A` 🔓

`lock_B` 🔒

`lock_B` 🔓

This slide has animations, use the 'slideshow' button to view it.

Solving deadlocks

- A simple rule to avoid deadlocks:
  - All thread *must* acquire locks in the same order
  - (also good if locks are released in reverse order, if possible)
- e.g., always acquire lock_A before lock_B

THREAD 1

1. acquire lock_A
2. acquire lock_B
3. do_somthing(A, B)
4. release lock_B
5. release lock_A

THREAD 2

1. acquire lock_A
2. acquire lock_B
3. do_somthing(A, B)
4. release lock_B
5. release lock_A

Atomics

- With hardware support, we can avoid data races without needing to use locks!
- In C, we can use ‘atomic types’, which guarantee that certain operations using them will be performed *atomically* (indivisibly) ⇒ no data race!
- Also avoids overhead of mutexes
- And since no locks are involved, we can’t introduce deadlock
- Atomics don’t solve all concurrency problems
- There are still some subtle problems (which we don’t cover in COMP1521)

Atomics

- Declaring an **atomic variable**
  - `atomic_int x = 10;`
  - `x += 1;` // Will be done atomically
  - `x = x + 1;` //Will **NOT** be done atomically!!!!
- Works with
  - ++, +=, - -, -=
  - |=, &=, ^=

Atomics

- Declaring an **atomic variable**
  - `atomic_int x = 10;`
  - `x += 1;` // Will be done atomically
  - `x = x + 1;` //Will **NOT** be done atomically!!!!
- A subset of functions in **stdatomic.h**:
  - atomic_fetch_add
    - `atomic_int x = 10;`
    - `int old = atomic_fetch_add(&x, 1);`
  - atomic_fetch_sub
  - atomic_fetch_or, atomic_fetch_xor, atomic_fetch_and

Add code with atomic in it

```
atomic_int bank_account = 0;


void *add_100000(void *argument) {
    for (int i = 0; i < 100000; i++) {
        // NOTE: This *cannot* be `bank_account = bank_account + 1`,
        // as that will not be atomic!
        // However, `bank_account++` would be okay
        // `atomic_fetch_add(&bank_account, 1)` would also be okay
        bank_account += 1;
    }
```

Concurrency is really complex!

- This is just a taste of concurrency!
- Other fun concurrency problems/concepts: livelock, starvation, thundering herd, memory ordering, semaphores, software transactional memory, user threads, fibers, etc.

- A number of courses at UNSW offer more:
  - COMP3231/COMP3891: [Extended] operating systems
  - COMP3151: Foundations of Concurrency
  - COMP6991: Solving Modern Programming Problems with Rust
  - … and more!

What we learnt Today

- Processes and Pipes
- Concurrency
- Threads
  - Data lifetime issues, Data races, deadlocks
  - Mutexes, atomics

Next Lecture

- Finish concurrency and threads
- Revision coding examples
  - Files, directories
  - Processes