Lecture 1 – Introduction to Concurrency

CS3211 Parallel and Concurrent Programming

Outline

- Concurrency vs. Parallelism
- Why should we understand concurrency?
- Enabling concurrency
- Problems in concurrent programs
- Hardware – OS – Compiler – Program

Concurrency – a definition

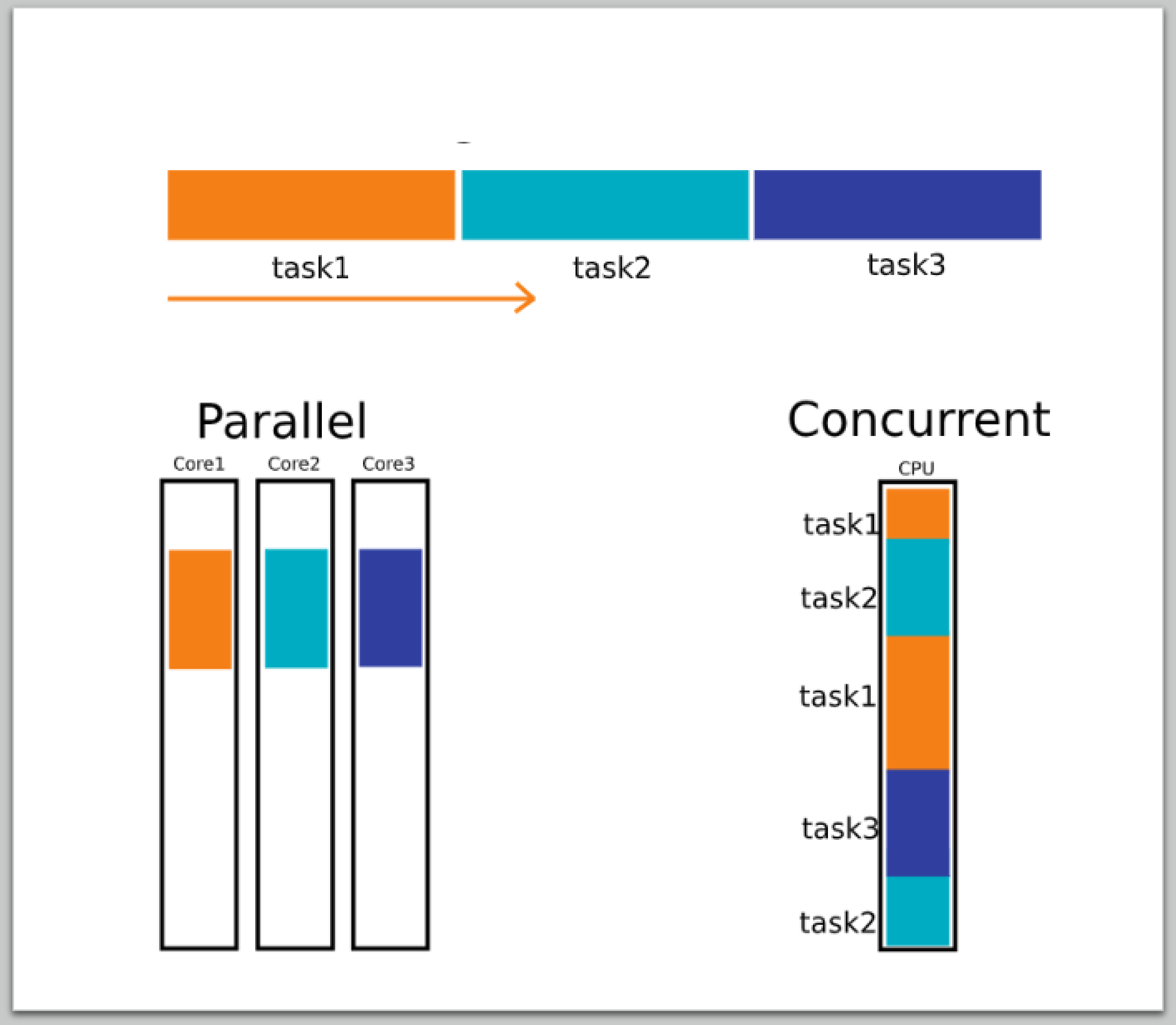
- Two or more separate activities happening at the same time
- Everywhere around us
  - Eating and watching the lecture
  - Listening and watching
- For computers, **concurrency** refers to a a single system performing multiple independent activities in parallel, rather than sequentially
  - Multitasking in OS

Why study concurrency?

- Not a new concept!
  - Traditionally concurrency was achieved through task switching

- Increased prevalence of computers that can genuinely run multiple tasks in parallel rather than just giving the illusion of doing so
  - *Illusion* of concurrency vs. *true* concurrency

An illusion?



task1
task2
task3
Parallel
Core1
Core2
Core3
Concurrent
CPU
task1
task2
task1
task3
task2

Concurrency vs. Parallelism

Concurrency

- Two or more tasks can start, run, and complete in overlapping time periods
- They might not be running (executing on CPU) at the same instant
- Two or more execution flows make progress at the same time by interleaving their executions or by executing instructions (on CPU) at exactly the same time

Parallelism

- Two or more tasks can run (execute) simultaneously, at the exact same time
- Tasks do not only make progress, but they also actually execute simultaneously

Hardware enables TRUE concurrency

- Computers are genuinely able to run more than one task in parallel
  - Multicore processors are used everywhere
  - Multiple processors used everywhere
  - High performance computing
- **Hardware threads** dictate the amount of TRUE concurrency
  - Processors have multiple cores
  - A core can support multiple hardware threads (SMT)

Enabling concurrency

- Multiple processes vs. multiple threads (safety vs. performance)

Process Interaction with OS

Exceptions

- Executing a machine level instruction can cause exception
- For example: Overflow, Underflow, Division by Zero, Illegal memory address, Mis-aligned memory access

Synchronous

  - Occur due to program execution

- Have to execute an exception handler

Interrupts

- External events can interrupt the execution of a program
- Usually hardware related: Timer, Mouse Movement, Keyboard Pressed etc.

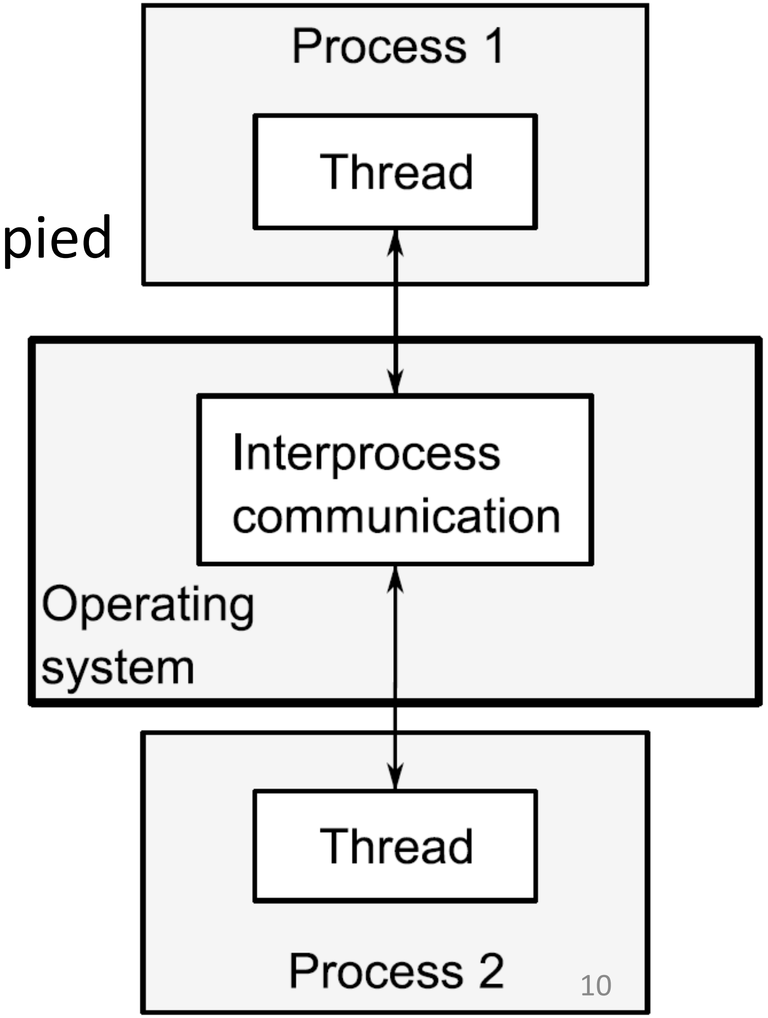
Asynchronous

  - Occur independently of program execution

- Have to execute an interrupt handler

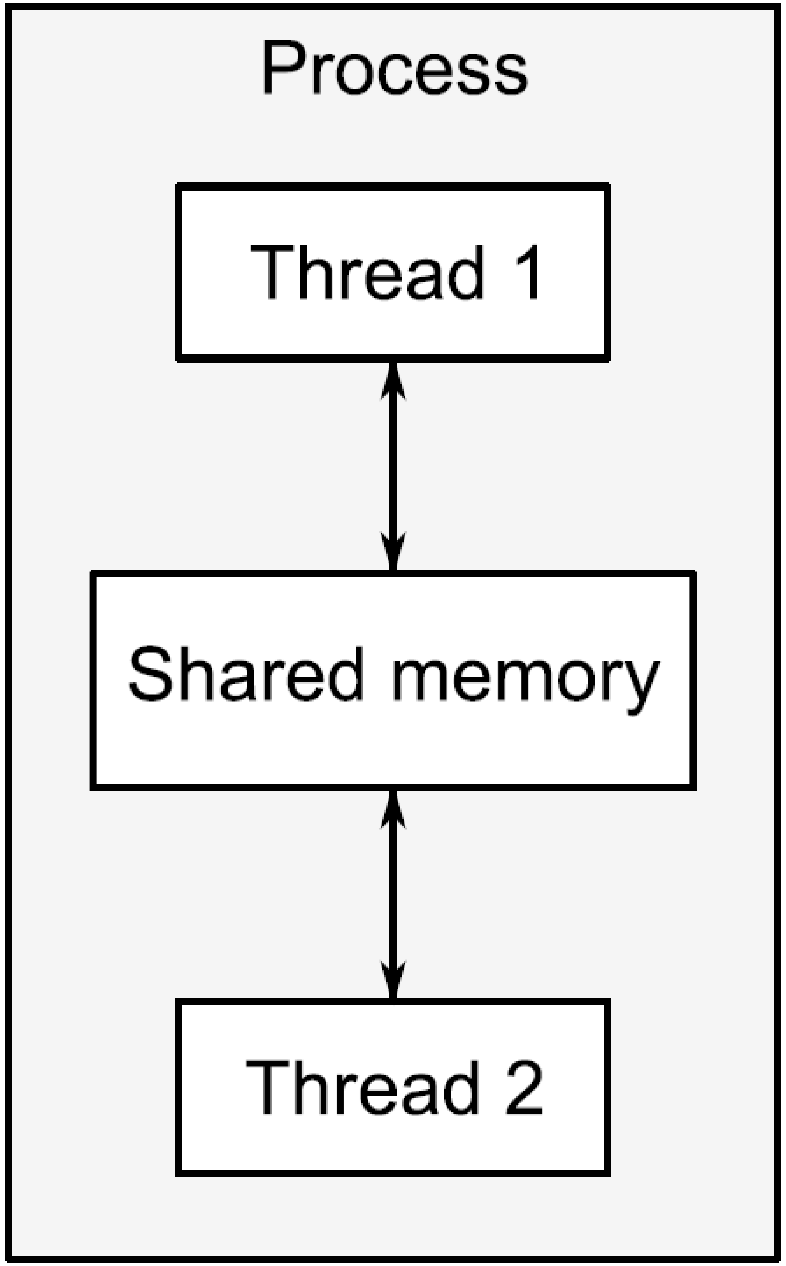
Disadvantages of processes

- Creating a new process is costly
  - Overhead of system calls
  - All data structures must be allocated, initialized and copied
- Communicating between processes costly
  - Communication enabled by the OS

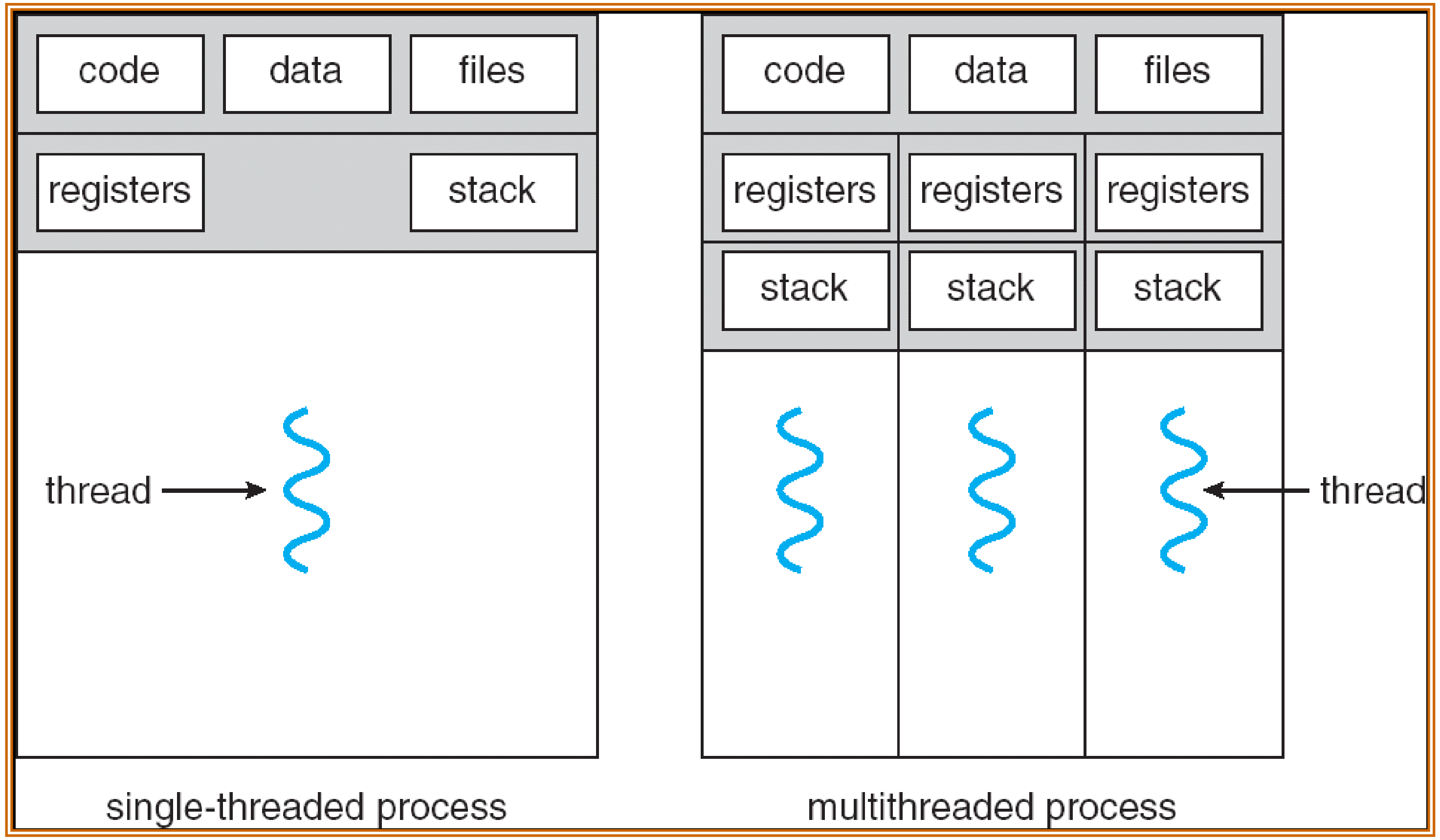


Threads

- Extension of process model:
  - A process may consist of multiple independent control flows called **threads**
  - The thread defines a sequential execution stream within a process(PC, SP, registers)
- Threads share the address space of the process



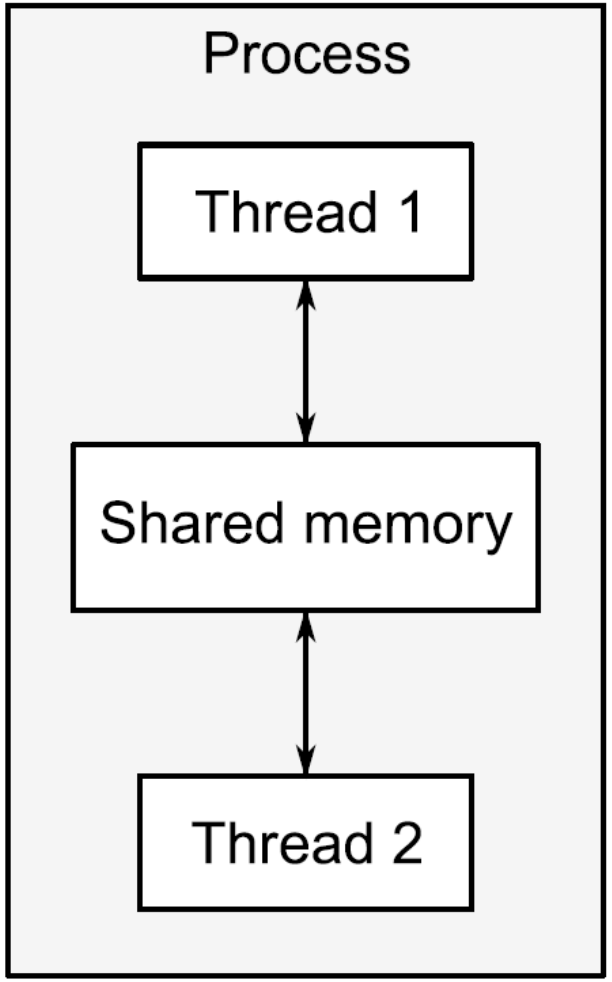
Process and thread: illustration



- Taken from Operating System Concepts (7th Edition) by Silberschatz, Galvin & Gagne, published by Wiley

Threads (cont.)

- Thread generation is faster than process generation
  - No copy of the address space is necessary
- Different threads of a process can be assigned run on different cores of a multicore processor
- 2 types of threads
  - User-level threads
  - Kernel threads



POSIX Threads

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
void *print_message_function( void *ptr );
main()
{
     pthread_t thread1, thread2;
     char *message1 = "Thread 1";
     char *message2 = "Thread 2";
     int  iret1, iret2;
    /* Create independent threads each of which will execute function */
     iret1 = pthread_create( &thread1, NULL, print_message_function, (void*) message1);
     iret2 = pthread_create( &thread2, NULL, print_message_function, (void*) message2);

     /* Wait till threads are complete before main continues. Unless we  */
     /* wait we run the risk of executing an exit which will terminate   */
     /* the process and all threads before the threads have completed.   */
     pthread_join( thread1, NULL);
     pthread_join( thread2, NULL);

     printf("Thread 1 returns: %d\n",iret1);
     printf("Thread 2 returns: %d\n",iret2);
     exit(0);
}
void *print_message_function( void *ptr )
{
     char *message;
     message = (char *) ptr;
     printf("%s \n", message);
}
```

Race condition

1. Two concurrent threads (or processes) access a shared resource without any synchronization

AND

2. At least one thread modifies the shared resource

- Solution: control access to these shared resources
- Necessary to synchronize access to any shared data structure
  - Buffers, queues, lists, hash tables, etc.

Mutual exclusion

- Use mutual exclusion to synchronize access to shared resources
  - This allows us to have large atomic blocks
- Code sequence that uses mutual exclusion is called critical section
  - Only one thread at a time can execute in the critical section
  - All other threads have to wait on entry
  - When a thread leaves a critical section, another can enter

Critical section requirements

1) Mutual exclusion (mutex)
   - If one thread is in the critical section, then no other is
2) Progress
   - If some thread T is not in the critical section, then T cannot prevent some other thread S from entering the critical section
   - A thread in the critical section will eventually leave it
3) Bounded waiting (no starvation)
   - If some thread T is waiting on the critical section, then T will eventually enter the critical section
4) Performance
   - The overhead of entering and exiting the critical section is small with respect to the work being done within it

Critical section requirements - details

- Requirements:
  - Safety property: nothing bad happens
  - Liveness property: something good happens
    - Progress, Bounded Waiting
  - Performance requirement
- Properties hold for each run, while performance depends on all the runs
- Rule of thumb: When designing a concurrent algorithm, worry about safety first (but don't forget liveness!)

Mechanisms

- Locks
  - Primitive, minimal semantics, used to build others
- Semaphores
  - Basic, easy to get the hang of, but hard to program with
- Monitors
  - High-level, requires language support, operations implicit
- Messages
  - Simple model of communication and synchronization based on atomic transfer of data across a channel
  - Direct application to distributed systems
  - Messages for synchronization are straightforward (once we see how the others work)

Deadlock

- Definition:
  - Deadlock exists occurs when the waiting process is still holding on to another resource that the first needs before it can finish.
- Deadlock is a problem that can arise:
  - When processes compete for access to limited resources
  - When processes are incorrectly synchronized

Condition for deadlock

Deadlock can exist **if and only if** the following four conditions hold simultaneously:

1. Mutual exclusion – At least one resource must be held in a non-sharable mode
2. Hold and wait – There must be one process holding one resource and waiting for another resource
3. No preemption – Resources cannot be preempted (critical sections cannot be aborted externally)
4. Circular wait – There must exist a set of processes [P1, P2, P3,…,Pn] such that P1 is waiting for P2, P2 for P3, etc.

Dealing with deadlock

- There are four approaches for dealing with deadlock:
  - Ignore it–how lucky do you feel?
  - Prevention–make it impossible for deadlock to happen
  - Avoidance–control allocation of resources
  - Detection and Recovery–look for a cycle in dependencies

Starvation

- Starvation is a situation where a process is prevented from making progress because some other process has the resource it requires
- Starvation is a side effect of the scheduling algorithm
  - OS: A high priority process always prevents a low priority process from running on the CPU
  - One thread always beats another when acquiring a lock

Disadvantages of concurrency

- Concurrency issues
- Maintenance difficulties
  - Complicated code
  - Debugging is challenging

- Threading overhead
  - Stack
  - Context switching

Concurrency benefits

- Separation of concerns
- Performance
  - Take advantage of the hardware
  - Optimization strategy

Types of parallelism

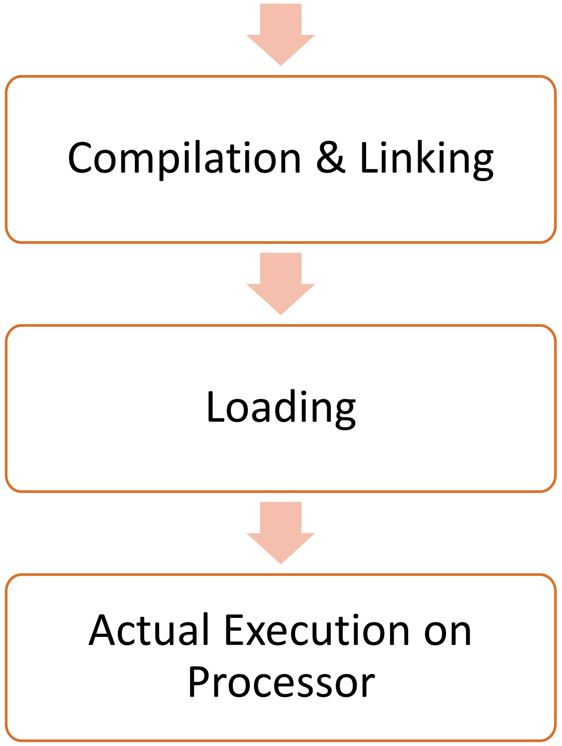
- Task parallelism
  - Do the same work faster
- Data parallelism
  - Embarrassingly parallel algorithms
  - Do more work in the same amount of time

Concurrent and Parallel Programming Challenges

- Finding enough parallelism (Amdahl's Law!)
- Granularity of tasks
- Locality
- Coordination and synchronization
- Debugging
- Performance monitoring

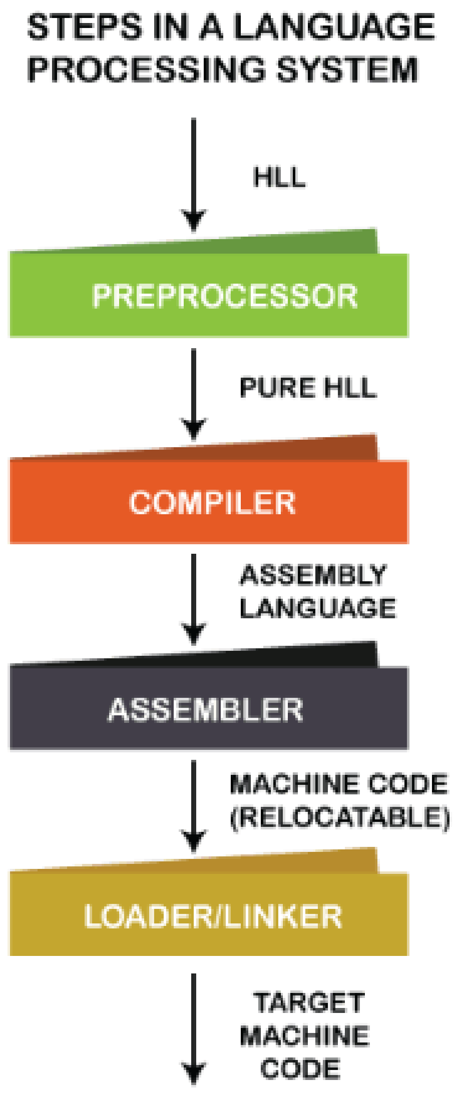
Execution of a (concurrent) program (C/C++)

- Compilation and linking
  - Done by the compiler
- Loading
  - The loader is usually specific to the OS
- Execution
  - Coordinated by the OS
  - The program gets access to CPU, memories, devices, etc.

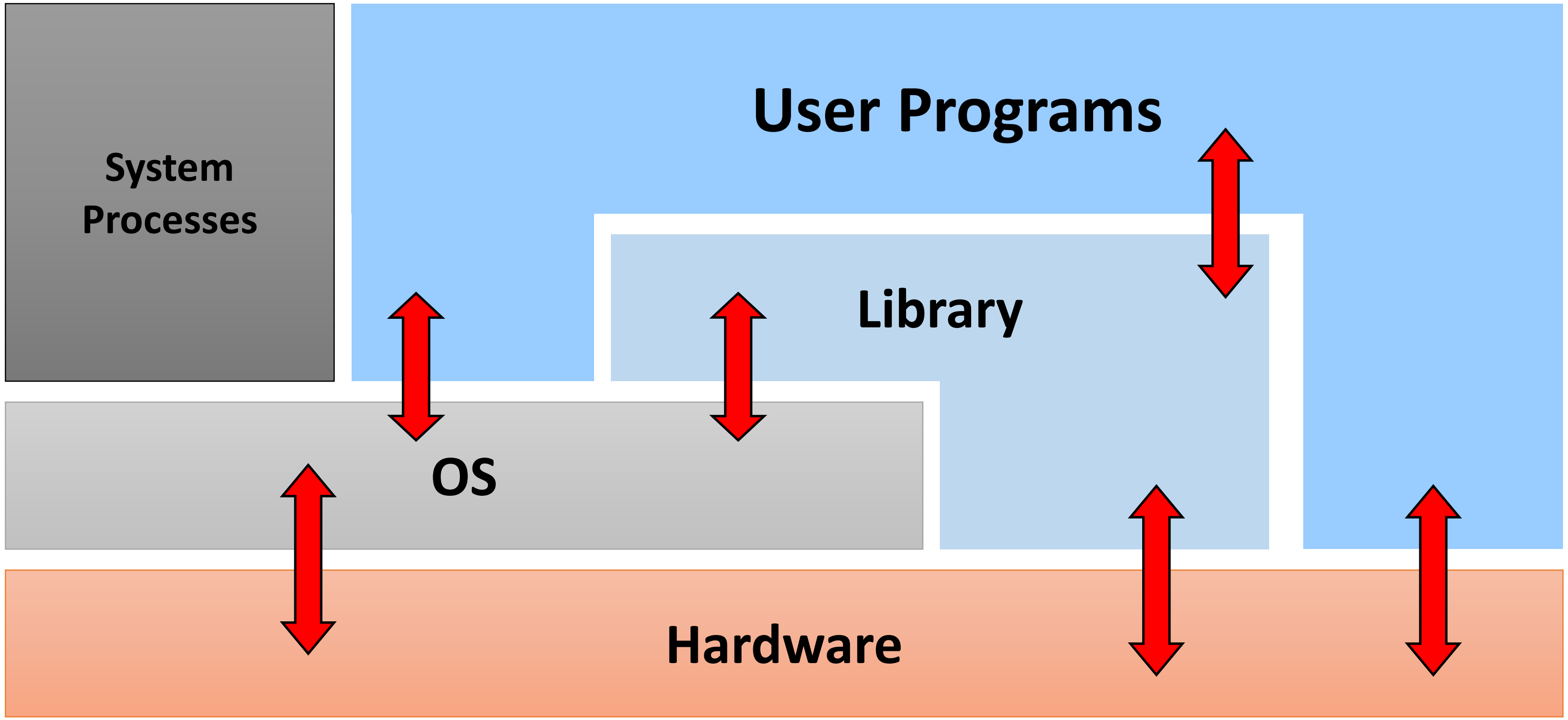


Building flow in C/C++

- Preprocessor
  - Replaces preprocessor directives (#include and #define)
- Compiler
  - Parses pure C++ source code and converts it into assembly
- Assembler
  - Assembles that code into machine code
- Linker
  - Produces the final compilation output from the object files the compiler produced



Where should we look at?



Summary

- Concurrency is our focus in this module
  - Assume concurrent programs run on an architecture that enables true concurrency (parallelism)
- Multithreading is commonly used to enable concurrency
  - Comes with many challenges and advantages