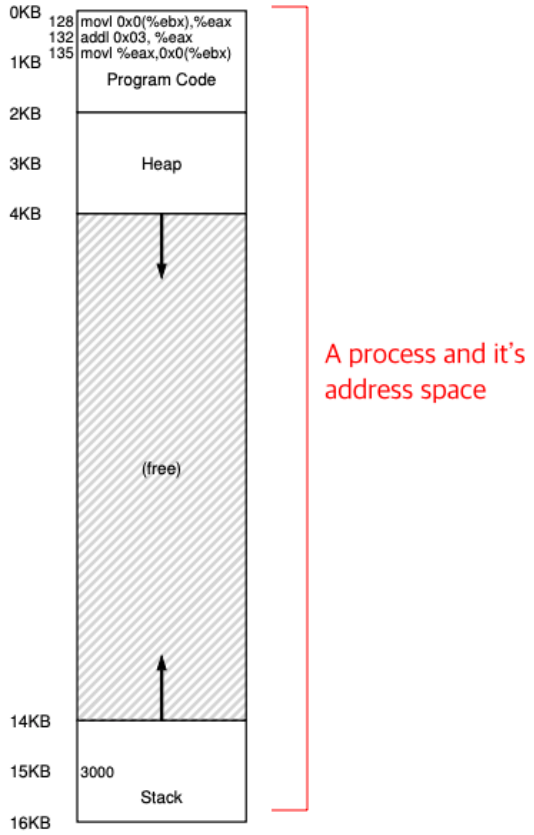
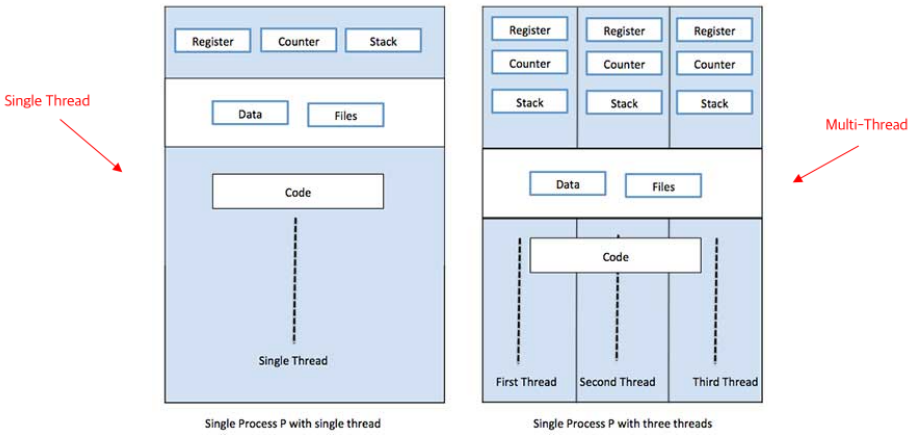
1 Concurrency

Vocabularies

1. **Address Space**
   - Is a range of discrete addresses where each corresponds to a memory cell



2. **Thread**
   - is the unit of execution within a process.
     - A process can have anywhere from one thread to many threads
3. **Multi-thread**
   - Is a process with more than one thread



4. **Context Switch**

   - is the process of storing the state of a process or thread, so that it can be restored and resume execution at a later point

5. **Process Control Block**

   - Is also called **process descriptor**
   - Is a data structure used by computer operating systems to store all the information about a process

```
// the registers xv6 will save and restore
// to stop and subsequently restart a process
struct context {
  int eip;
  int esp;
  int ebx;
  int ecx;
  int edx;
  int esi;
  int edi;
  int ebp;
};

// the different states a process can be in
enum proc_state { UNUSED, EMBRYO, SLEEPING,
                  RUNNABLE, RUNNING, ZOMBIE };

// the information xv6 tracks about each process
// including its register context and state
struct proc {
  char *mem;                  // Start of process memory
  uint sz;                    // Size of process memory
  char *kstack;               // Bottom of kernel stack
                              // for this process
  enum proc_state state;      // Process state
  int pid;                    // Process ID
  struct proc *parent;        // Parent process
  void *chan;                 // If !zero, sleeping on chan
  int killed;                 // If !zero, has been killed
  struct file *ofile[NOFILE]; // Open files
  struct inode *cwd;          // Current directory
  struct context context;     // Switch here to run process
  struct trapframe *tf;       // Trap frame for the
                              // current interrupt
};
```

register context data structure

Process Control Block

Where register context Is

6. **Thread Control Block**

- Is also called **process descriptor**
- Is a data structure used by computer operating systems to store all the information about a thread

7. **Parallelism**
   - Is a condition that arises when at least two threads are executing simultaneously.
8. **Parallelization**
   - Is the task of transforming a standard **single-threaded** program into a program that does this (sort of) work on multiple CPUs
9. **Multiprogramming**
   - Is a computer running more than one program at a time (like running Excel and Firefox simultaneously).
10. **Critical Section**
    - Is a piece of code that acceses a shared resource, usually a variable or data structure
11. **Race Condition (Data race)**
    - Is a condition that arises if multiple threads of execution enter the critical section at roughly the same time; both attempt to update the shared data structure, leading to a suprising outcome
12. **Mutual Exclusion**
    - Is a concurrency control property which is introduced to prevent race conditions, resulting in deterministic program output.
13. **Deterministic Execution**
    - Means path of execution is fully determined by the specification of computation
    - Is guaranteed to procduce the same outcome, given the same input
14. **Indeterminate**
    - Means path of execution isn't fully determined by the specification of computation
    - Same input can produce different outcomes
15. **Synchronization Primitives**

- Are simple software mechanisms provided by a platform (e.g. operating system) to its users for the purposes of supporting thread or process synchronization.

  Example

  Mutex, event, conditional variables and semaphores

16. **Atomic**
    - Means "All or nothing"; no in-between state
17. **Transaction**
    - is the grouping of many actions into a single atomic action
18. **Conditional Variable**
    - Is a synchronization primitives that enable threads to wait until a particular condition occurs

1.1 Why Use Threads?

- **Parallelism**
  - Potential of speeding up a process by using multiple processors
    - by making 1 CPU $\rightarrow$ perform a portion of work
  - **Parallelization** converts **single-threaded** programs to (sort of) **multi-threaded** program
- Avoid blocking program due to slow I/O
  - Switch to another thread while a thread waits (in blocked state) for I/O
  - **Thread** is a natural way of getting unstuck
    - Enables overlap of I/O within a single program

    Example

    - Typing on Messenger (thread A) while (thread B) is sending 'sent' message to user
    - Typing on Visual Studio Code (thread A), while (thread B) is saving data to hard drive

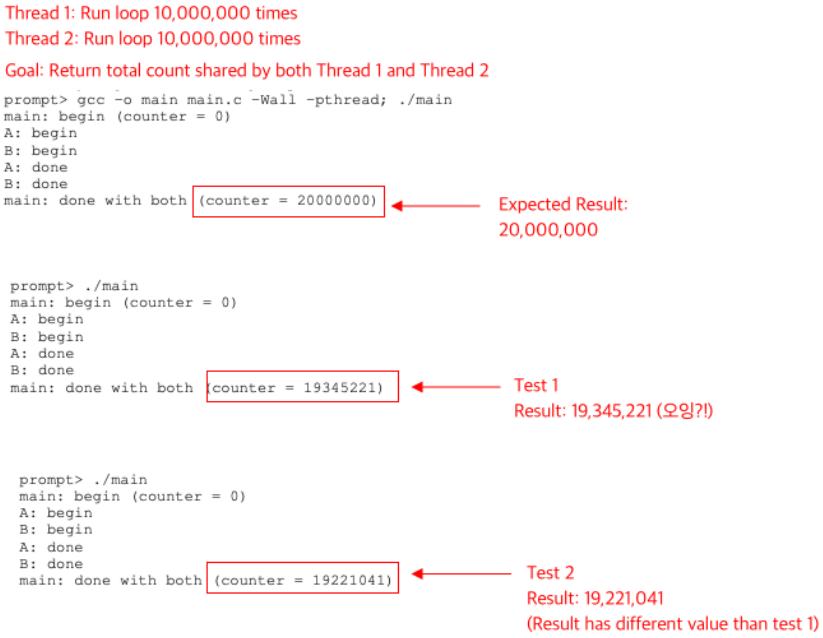
1.2 Thread Creation

- Created using function `pthread_create`
- Created thread is in one of two states

1. Ready (wait until something happens)
2. Running (run immediately)

1.3 Why It Gets Worse: Shared Data

- **Thread** doesn't produce **deterministic** result



1.4 The Heart of The Problem: Uncontrolled Scheduling

- **Race Condition (Data race)**
  - Two threads accessed the shared variable at the same time
  - Should increase by 2, but only 1 occured
- Shared variable is a **critical section**
- **Mutual Exclusion** is needed

1.5 The Wish For Atomicity

- **Mutual Exclusion** guarentees **atomicity** in **critical section**
- Is done using **synchronization primitives**
  - Allows multi-threaded code to access **critical section** in a synchronized and controlled manner and produce correct result

Examples

Mutex, event, conditional variables and semaphores

1.6 One More Problem: Waiting For Another

- There are two common problems in concurrency
  1. Accessing a shared variable
  2. One thread waiting for another to complete some action before it can continue
     - Is solved by **conditional variable**

     Example

     Put a process to sleep when performing disk I/O; when I/O is complete, wake up from sleep and continue