



•
5 0 . 0 0 5 C S E

Natalie Agus
Information Systems Technology and Design
SUTD

MAKING PROGRAM

```
1 //
2 // cKnowledge.c
3 // CSE_CCodes
4 //
5 // Created by Natalie Agus on 14/1/19.
6 // Copyright © 2019 Natalie Agus. All rights reserved.
7 //
8
9 #include "cKnowledge.h"
10
11
12 void func1(void) { printf("Function 1 Called\n"); }
13 void func2(void) { printf("Function 2 Called\n"); }
14 void func3(void) { printf("Function 3 Called\n"); }
15
16 int testFunctionPointers(void)
17 {
18     static void (*ptr[3])(void) = {func1, func2, func3};
19     int k=0;
20     for(k=0; k<3; k++)
21         ptr[k]();
22     return 0;
23 }
```

Your code
in plaintext format, commonly

Compiled



A program, it is executable
A set of instructions, that can tell the kernel
what to do to create a **process image**

Process image: everything about a process at a
point of time, its heap, stack, registers value,
data, instructions, as well as its PCB data
structure

MAKING PROCESS



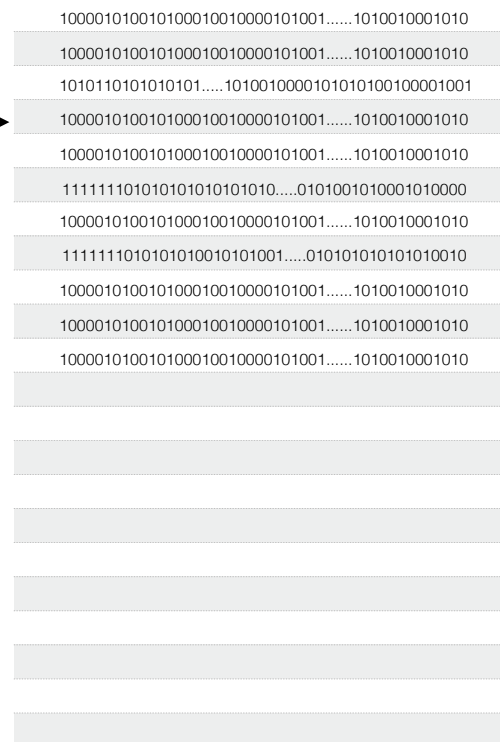
A program, executable

A set of instructions, that can tell the kernel what to do to create a **process image**

A program doesn't change over time.
It is passive in nature.

Executed

PC →



Virtual Memory

A **process image** from the executable is loaded into the **private** VM space of the process (can be on RAM or disk). This gives the initial state of a process.

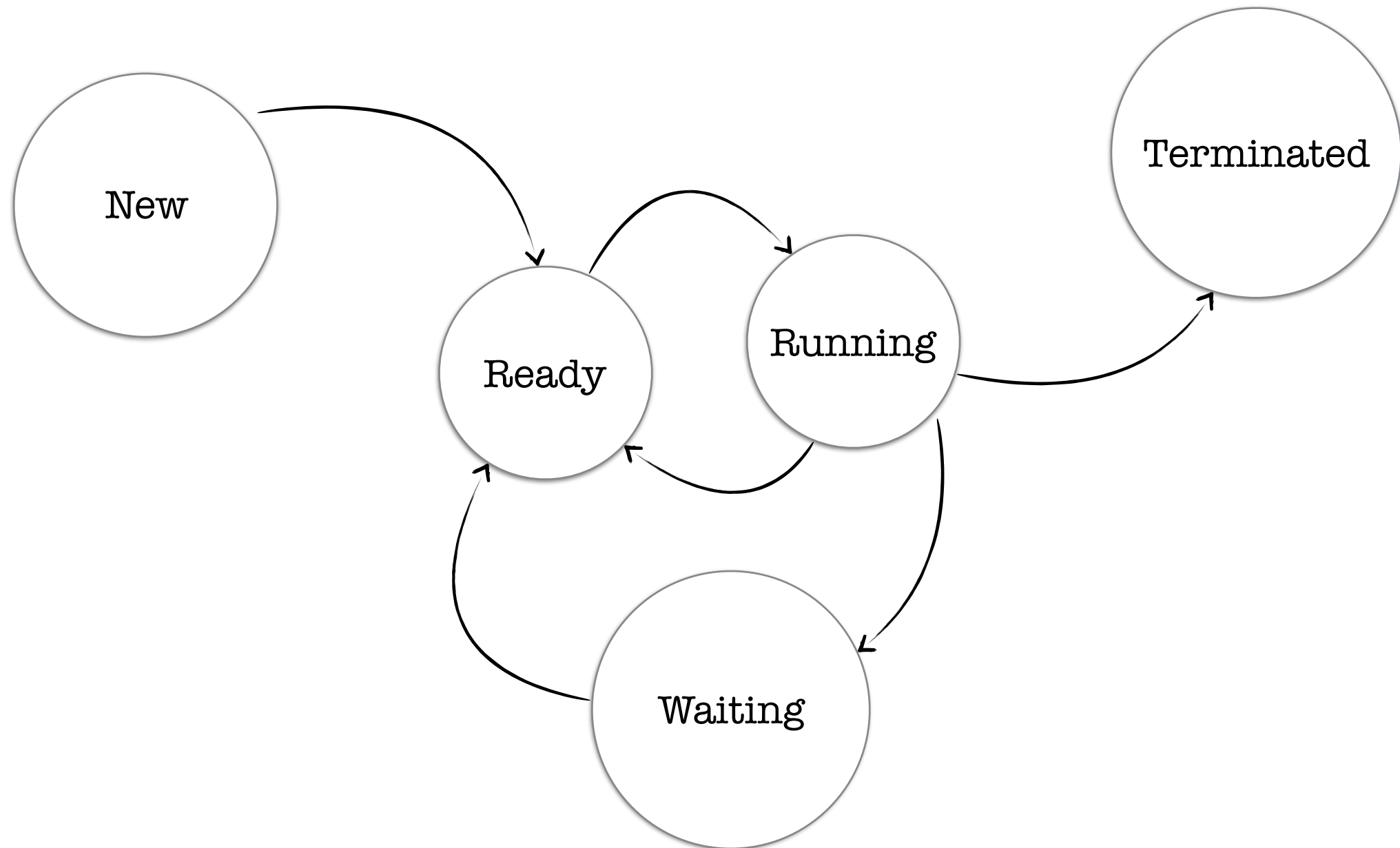
Also, an entry in the process table is created to indicate that a new process is going to be run in the system.

Then as the PC moves and execute the instructions, **a process is happening.**

A process changes over time.
It is active in nature.

•

SCHEDULING STATE TRANSITION



•

PROCESS CONTROL BLOCK

A *data structure* that stores information about each process **for scheduling purposes**.

The OS scheduler manage the scheduling of processes, and keeps track of all the processes' PCB in the process table

```
pid_t pid; /* process identifier */
long state; /* state of the process */
unsigned int time_slice /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files; /* list of open files */
struct mm_struct *mm; /* address space of this process */
```

struct task_struct

Other typical process information:

1. Process state
2. Program counters
3. CPU registers
4. CPU scheduling information (priority, scheduling protocol)
5. Memory management: pagetable, value of base and limit regs
6. I/O: list of open files, connected devices
7. Accounting: time running, resources used

•

THE PROCESS TABLE

0	0	2	1	0	0	1
P1	P2	P3	P4	P5	P6	P7



Context switch

•

P R O C E S S S C H E D U L I N G Q U E U E S

Three main types of queue:

1. Job queue : **set of all processes in the system**
2. Ready queue : **resides in RAM, ready to exec**
3. Device queue : **wait for I/O devices**

Processes may migrate among different types of queue

C O N T E X T S W I T C H

Interleave executions between processes in the system

Is an **overhead** because switching does no useful work apart from giving the "**concurrency**" illusion for single core CPUs

SCHEDULING

INTERLEAVED EXECUTION

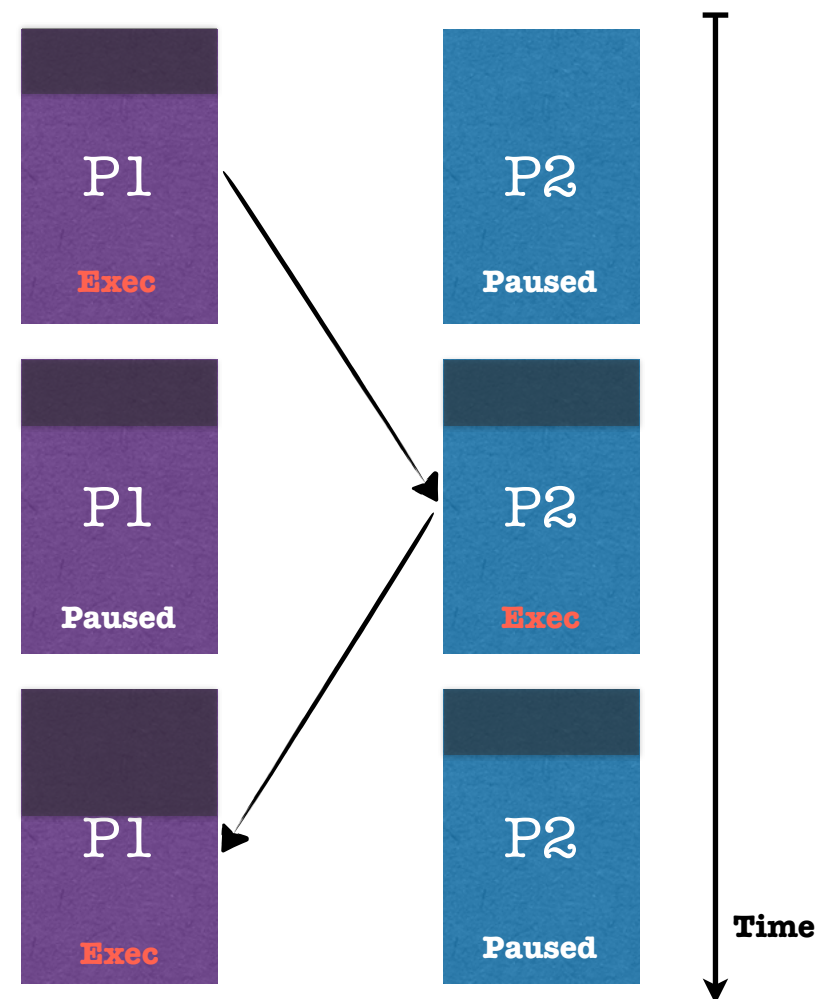
Processes take turns to be executed bit by bit

CONCURRENCY

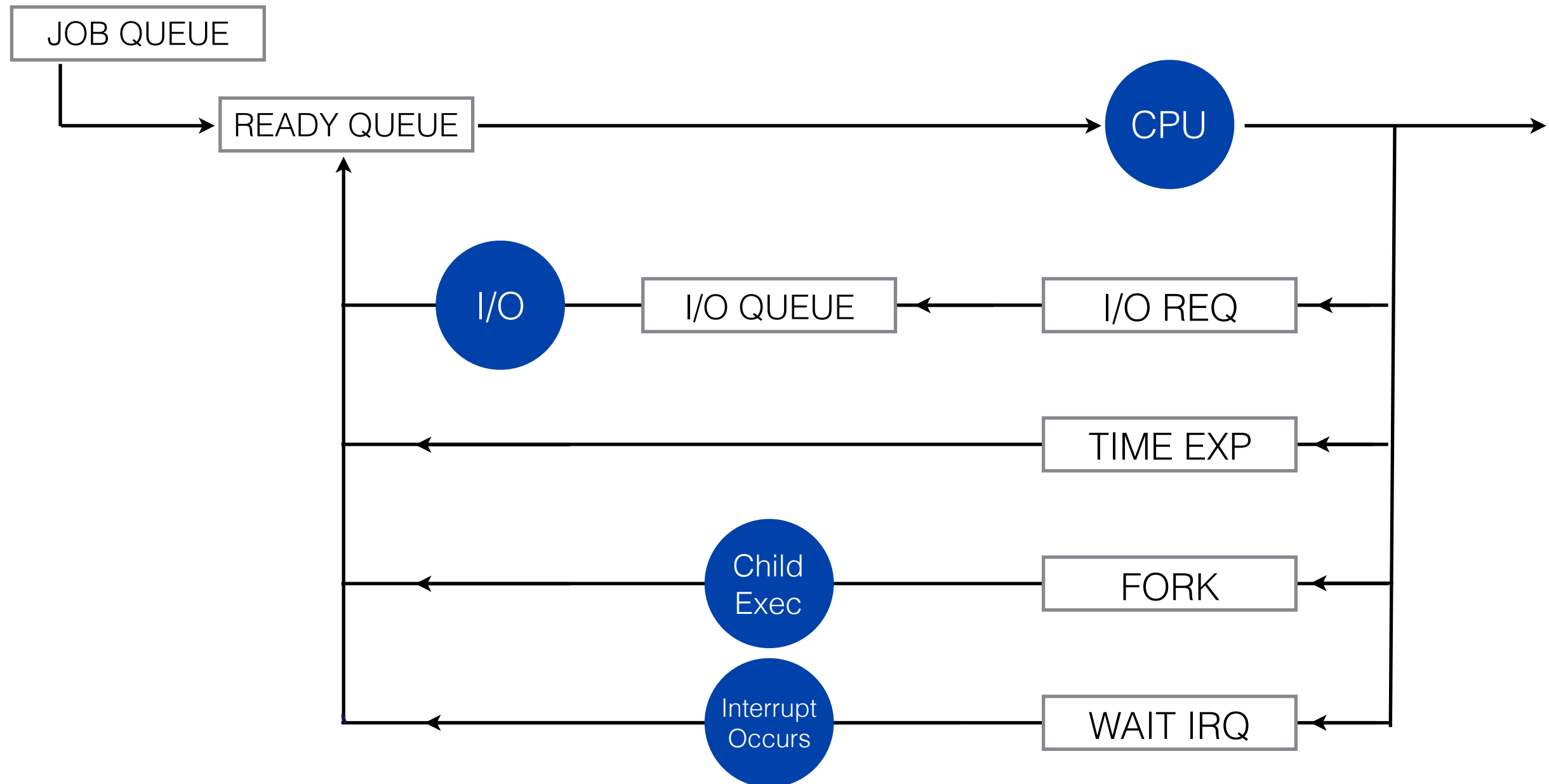
Interleaving process execution gives the illusion of concurrency, where all the programs seem to "run" at the same time

PARALLELISM

Only multi-core systems (multiple CPU) achieves true parallelism

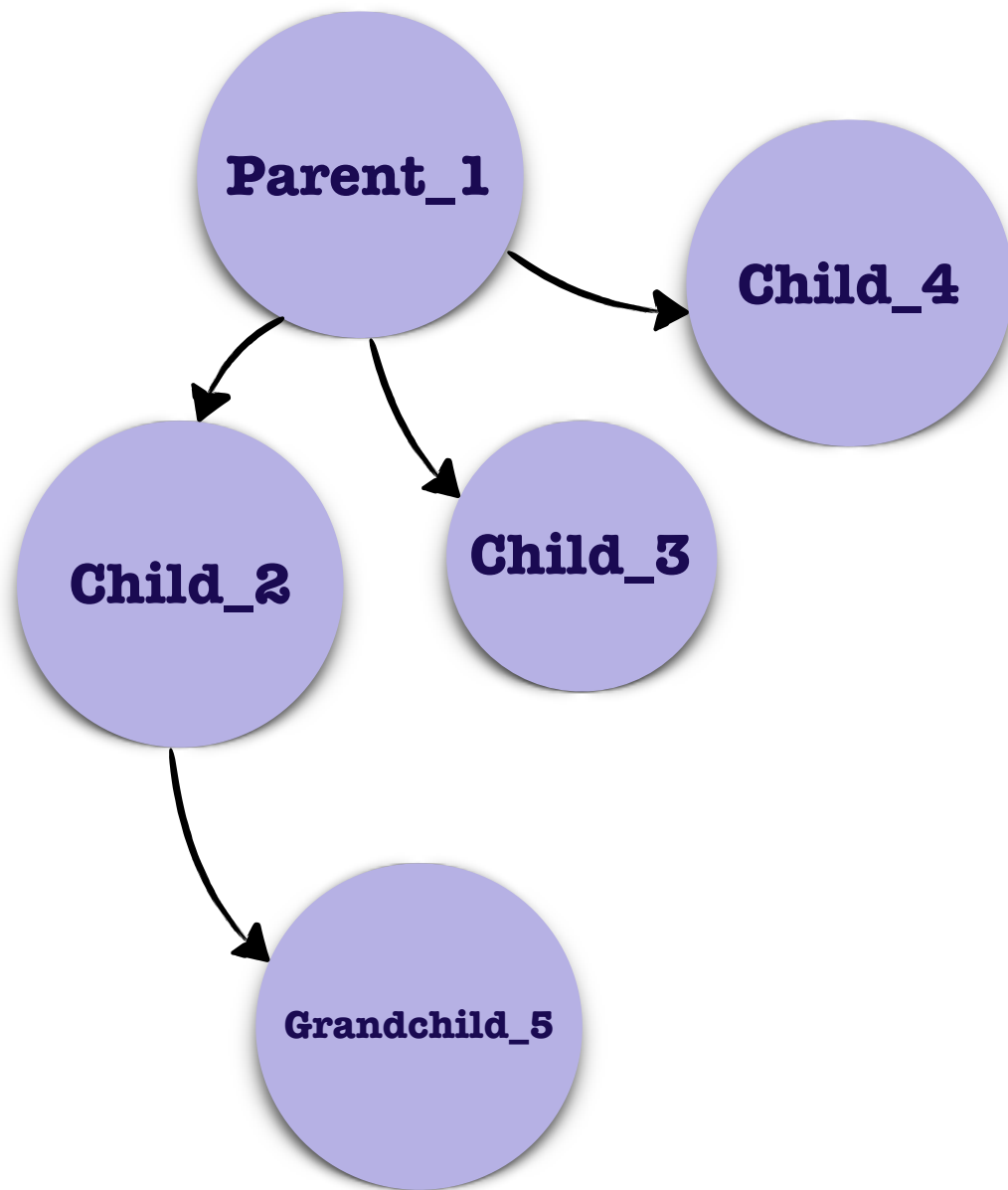


PROCESS QUEUING DIAGRAM



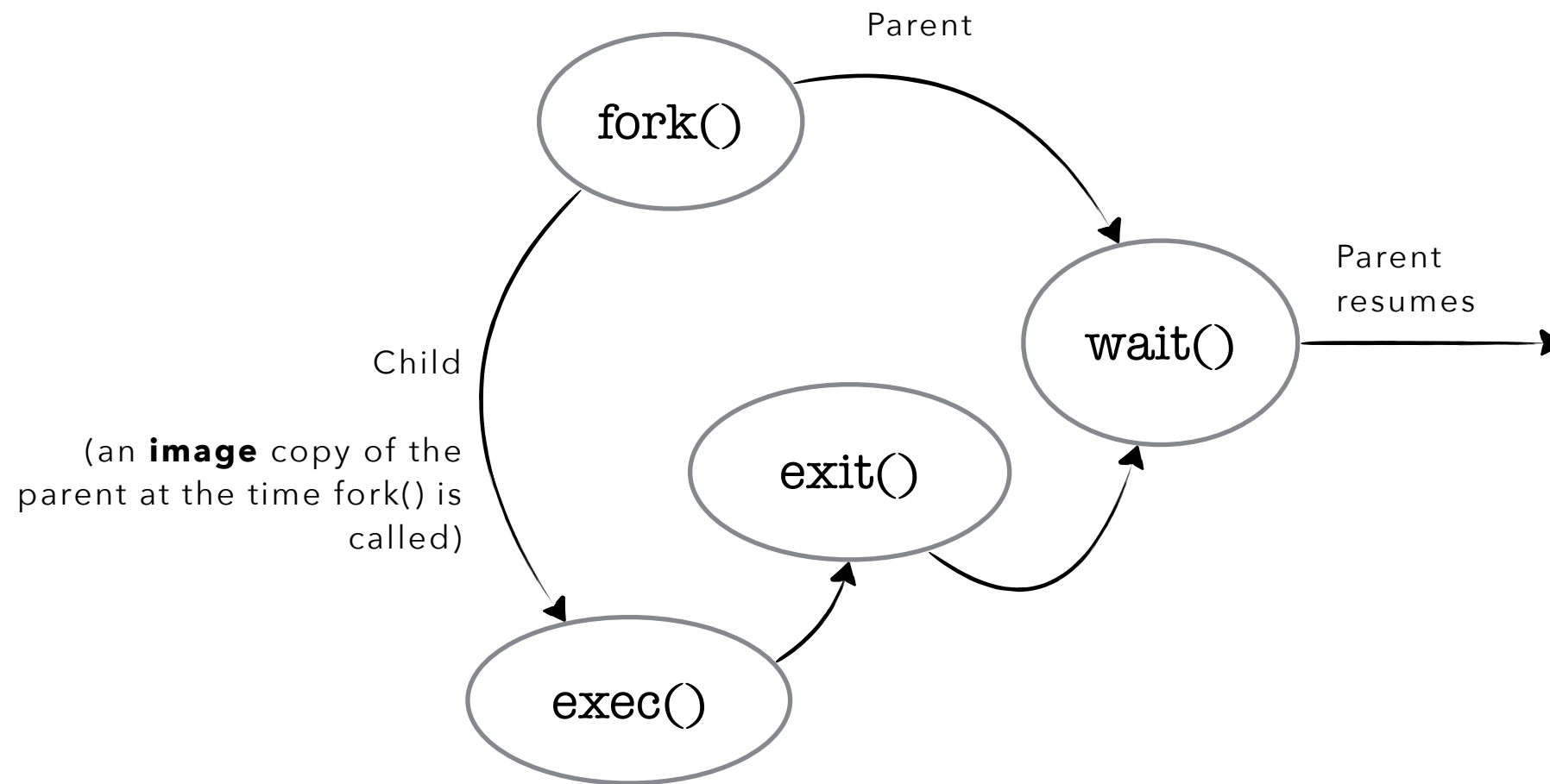
PROCESS CREATION

New processes are created by **fork()** system call in UNIX-based machines



- Each process can create another process, forming process tree.
- Processes can execute **concurrently**
- Parents and children belong to **different virtual address space**, but they can share resources and communicate as well using **shared memory**
- Children is a **duplicate** of parent upon creation
- Parents can wait for children using `wait()`, but not the other way around

• FORK() SYSTEM CALL



1. `exec()` replaces the current process image with a new process image.
2. Loads the program into the new process space and runs it from the **entry** point

• FORK SYSTEM CALL IN C

```
void fork_example_3(){  
    int returnValue = fork();  
    // child process because return value zero  
    if (returnValue == 0){  
        printf("Hello from Child!\n");  
        exit(0);  
    }  
  
    // parent process because return value non-zero.  
    else{  
        printf("Hello from Parent!\n");  
        wait();  
    }  
    //parent or child can run concurrently  
    //the order of output is unclear, because  
    //we don't know how OS execute them  
}
```

Output possibility 1:

Hello from Child!
Hello from Parent!

Output possibility 2:

Hello from Parent!
Hello from Child!

```

void fork_example_3(){

    int returnValue = fork();
    // child process because return value zero
    if (returnValue == 0){
        printf("Hello from Child!\n");
        exit(0);
    }

    // parent process because return value non-zero.
    else{
        printf("Hello from Parent!\n");
        wait();
    }
    //parent or child can run concurrently
    //the order of output is unclear, because
    //we don't know how OS execute them
}

```

Parent

Child

```

void fork_example_3(){

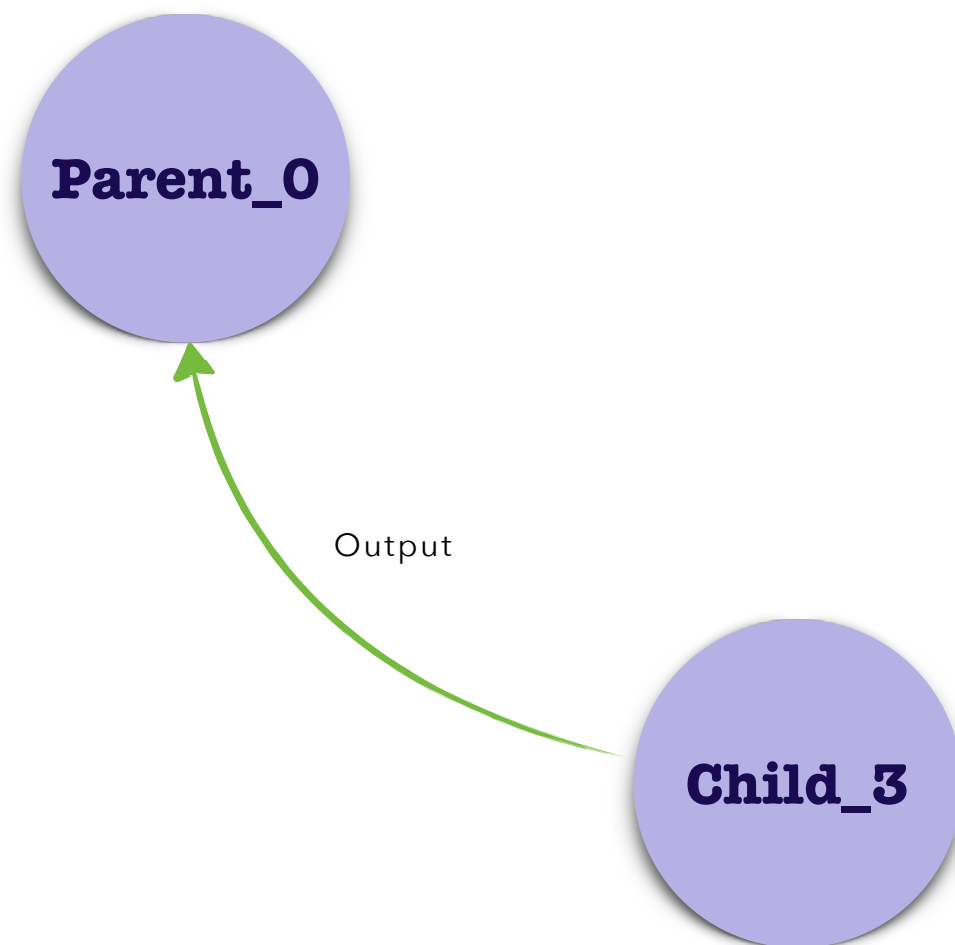
    int returnValue = fork(); —Entry point
    // child process because return value zero
    if (returnValue == 0){
        printf("Hello from Child!\n");
        exit(0);
    }

    // parent process because return value non-zero.
    else{
        printf("Hello from Parent!\n");
        wait();
    }
    //parent or child can run concurrently
    //the order of output is unclear, because
    //we don't know how OS execute them
}

```

PROCESS TERMINATION

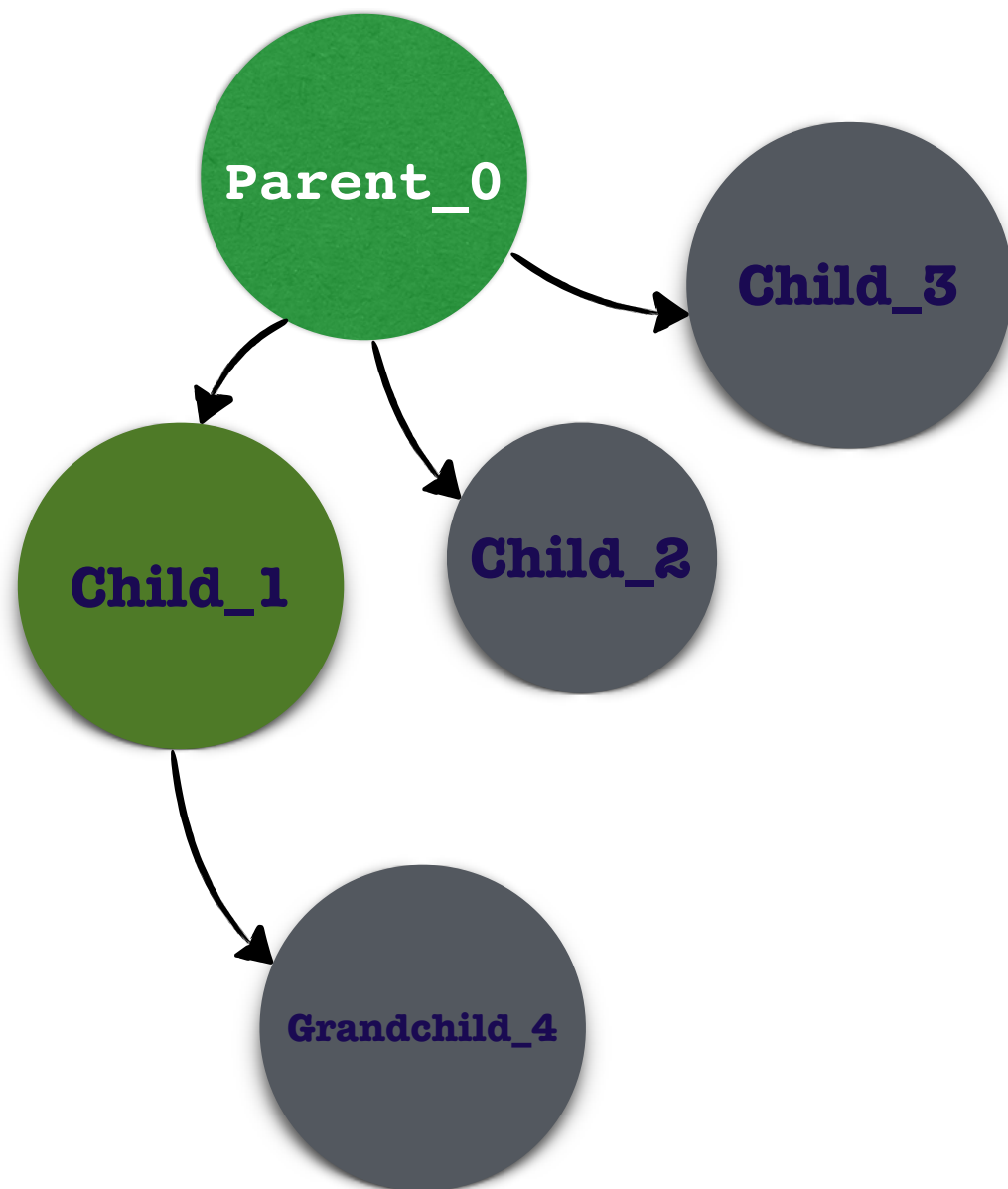
Done via **exit()** system call. Resources will be deallocated by the OS



- Parent processes can abort child processes
 - If child exceeds resources (memory /time)
 - If task is no longer required
 - If parent wants to exit
- Child process can exit using system call **exit(status)**
- Usually **status** is an int indicating whether or not the exiting process is successful
- Parent process can receive output from child process when it waits for child process to exit: **pid_t wait(&status)**

ZOMBIE PROCESSES

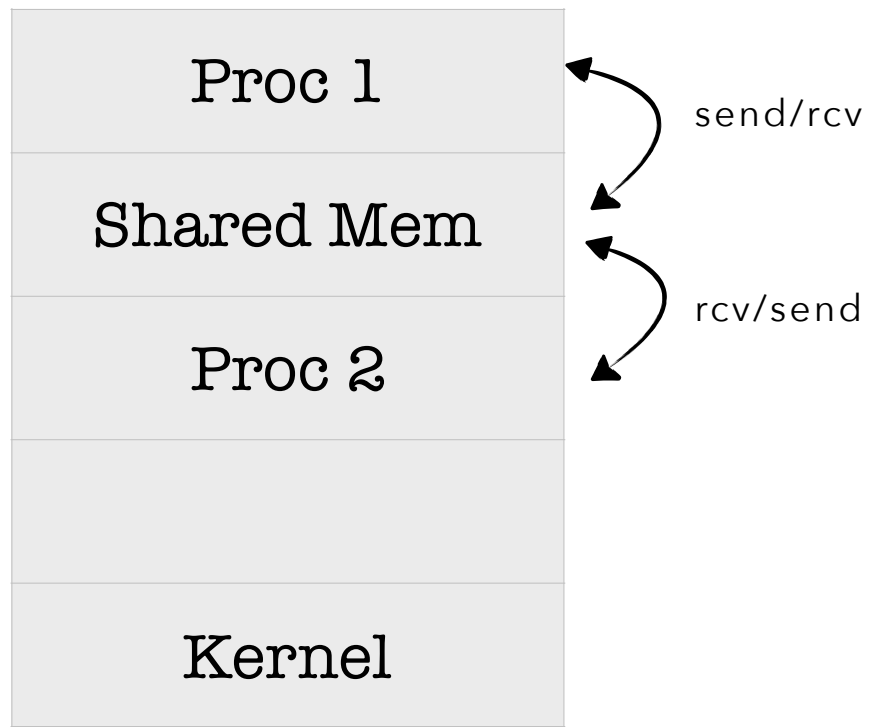
Child processes who are already terminated become zombie processes until their parents read their exit status



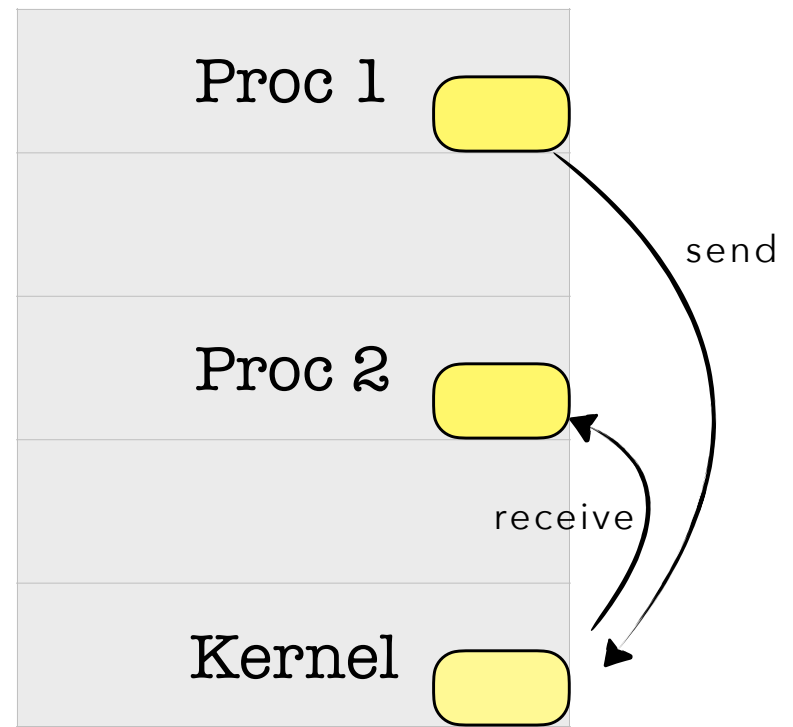
- The grey nodes: processes have terminated, but parents (in green) continue execution and does not call wait()
- All the children processes exit status is not reaped, hence turning into **zombie** processes
- OS can help abort **zombie** processes by themselves
- Otherwise, they will remain as a PCB entry in the proctable until the parent terminates / computer restarts

•

I N T E R P R O C E S S E S C O M M U N I C A T I O N



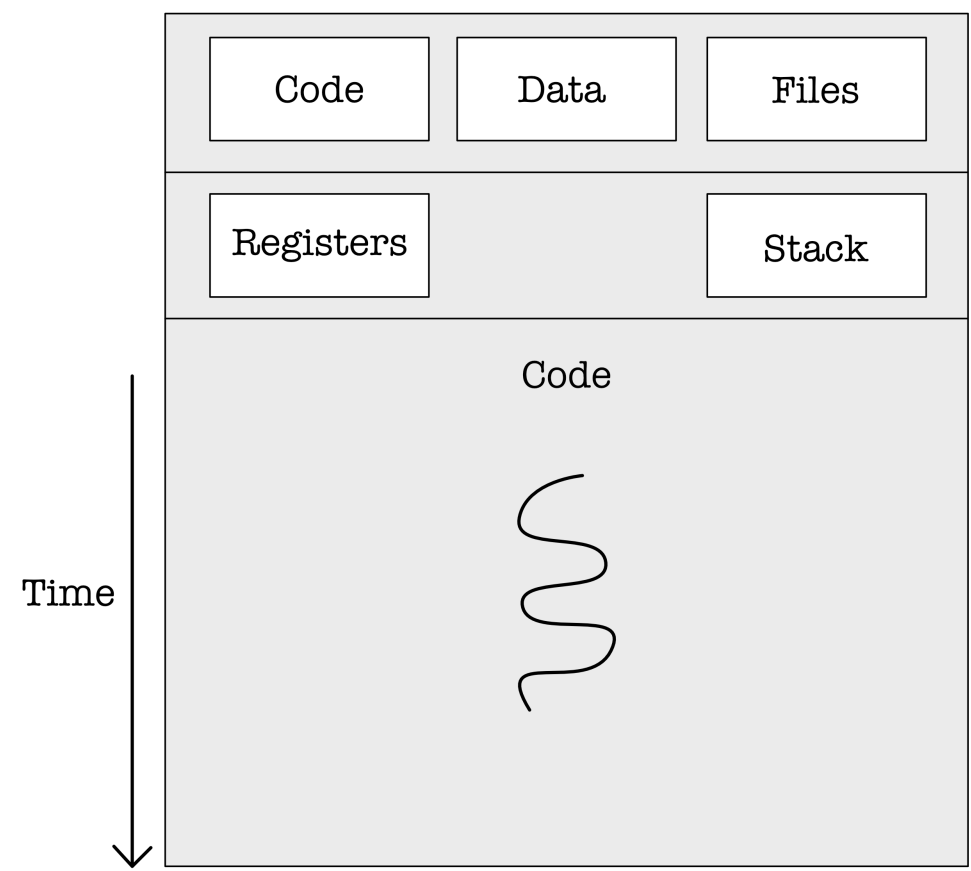
Shared memory



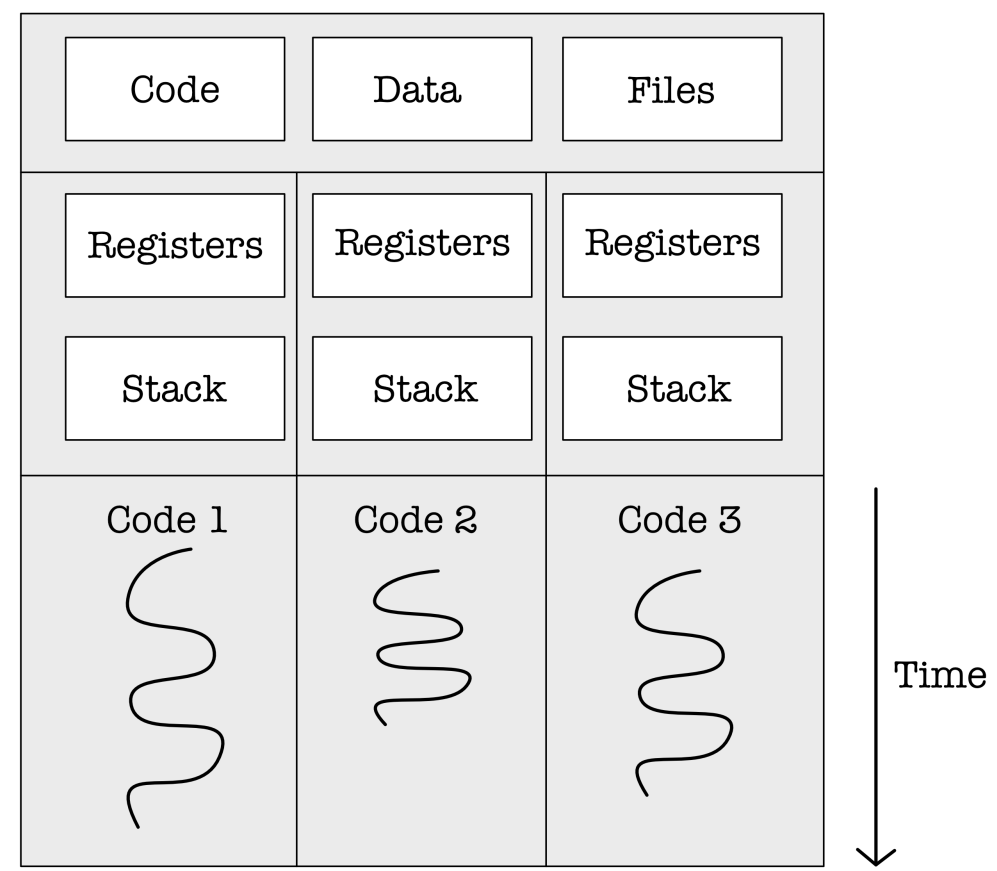
Sockets

•

T H R E A D S



A single process with
a single thread



A single process with
three threads

PROCESSES

More overhead to create

Requires private address space, has protection against other processes

Processes are independent of one another, no concurrency issues

No synchronization overhead, easier to program and work as intended

Can benefit from parallel execution on multiprocessor system

A process can have many threads executing different tasks

•

THREADS

Lesser overhead to create

Easier to create since they share address space, only differ in program execution

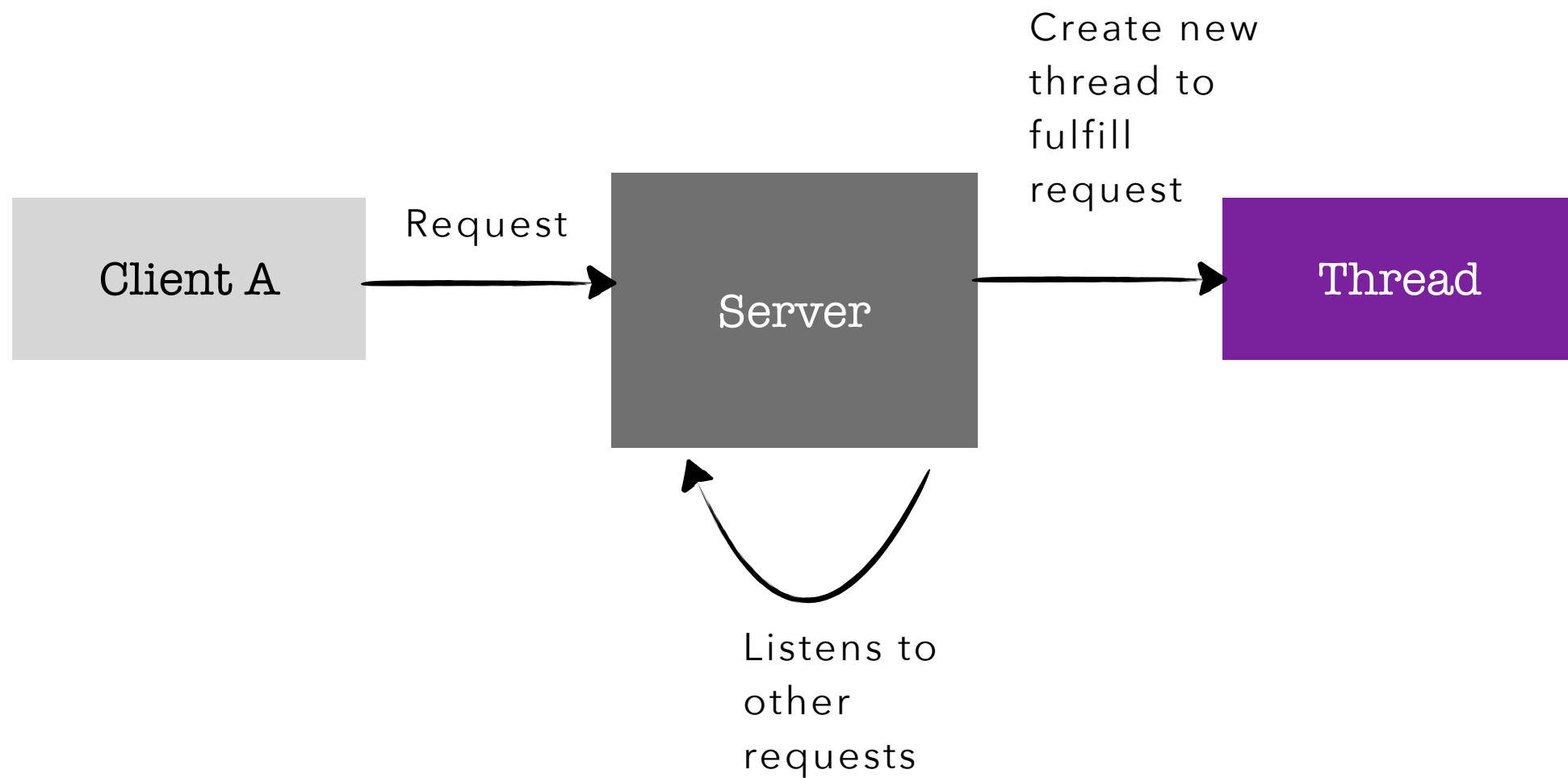
Requires careful programming since threads of the same process share data structures and hence are interdependent

Can potentially suffer from synchronization overheads, harder to program and work as intended

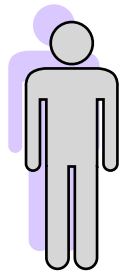
Can benefit from parallel execution on multiprocessor system

Good for responsiveness of a process

MULTITHREADED ARCHITECTURE

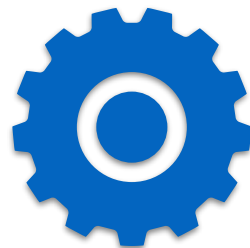


TYPES OF THREADS



User threads

- Not known to kernel
- Scheduled by thread scheduler in thread library
- Runs in user mode, cheaper to create



Kernel threads

- Has kernel data structure: thread control block (expensive to create and context switch)
- Scheduled and known by kernel

•

THREAD MAPPING

