Processes and THREADS

50.005 CSE

Natalie Agus
Information Systems Technology and Design
SUTD

MAKING PROGRAM

```c
//
//  cKnowledge.c
//  CSE_CCodes
//
//  Created by Natalie Agus on 14/1/19.
//  Copyright © 2019 Natalie Agus. All rights reserved.
//

#include "cKnowledge.h"


void func1(void) {  printf("Function 1 Called\n"); }
void func2(void) {  printf("Function 2 Called\n"); }
void func3(void) {  printf("Function 3 Called\n"); }

int testFunctionPointers(void)
{
    static void (*ptr[3])(void) = {func1,func2,func3};
    int k=0;
    for(k=0;k<3;k++)
        ptr[k]();
    return 0;
}
```

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**

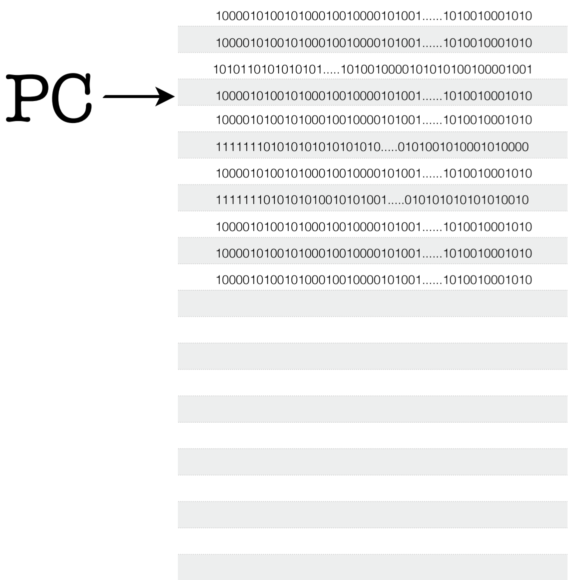
Process image: everything about a process at a point of time, its heap, stack, registers value, data, instructions, as well as its PCB data structure

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



Virtual Memory

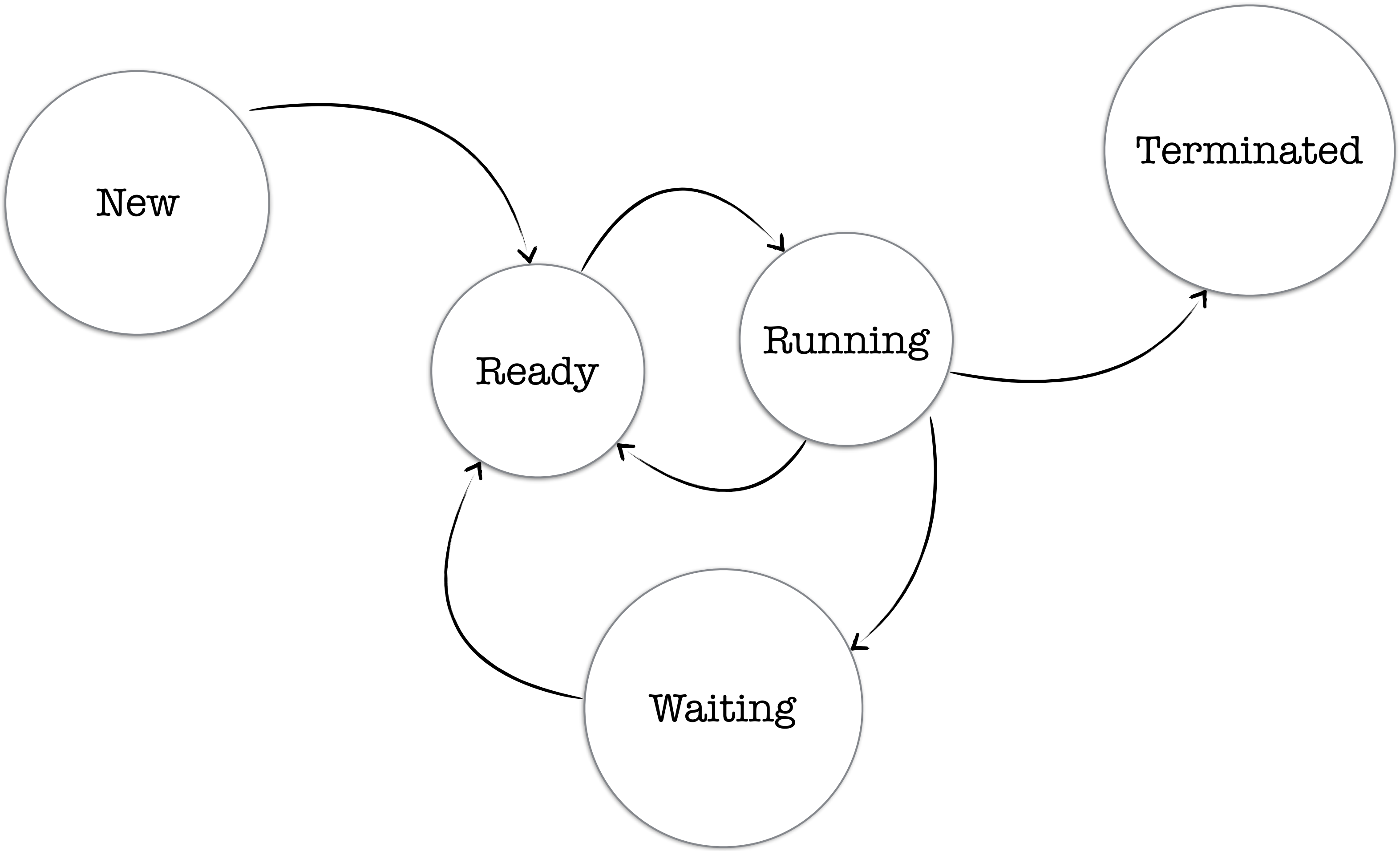
A **process image** from the executable is loaded into the **private** VM space of the process (can be on RAM or disk). This gives the initial state of a process.

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*

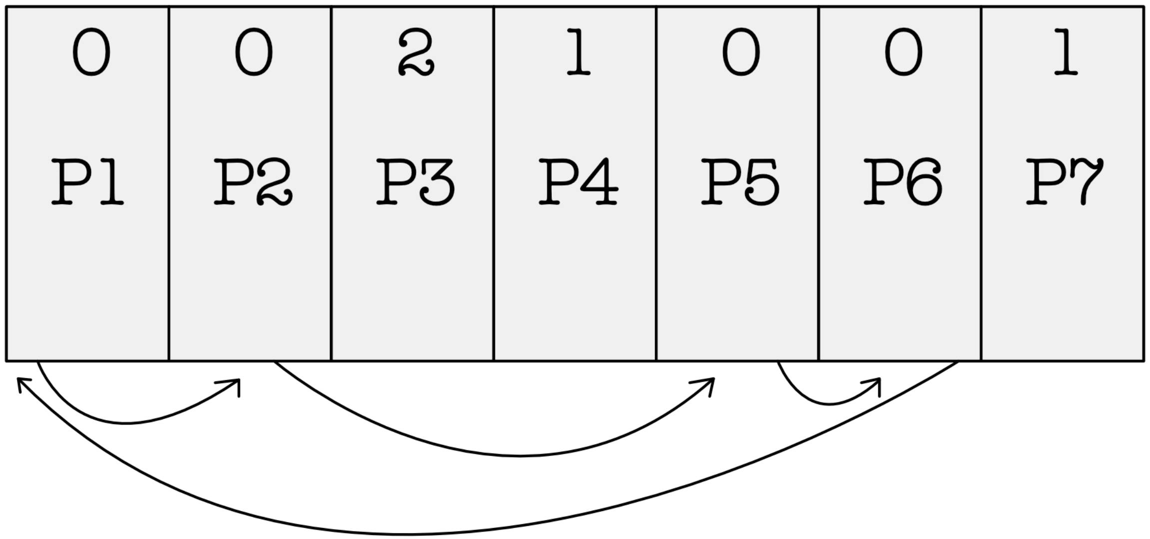
```
pid_t pid; /* process identifier */
long state; /* state of the process */
unsigned int time_slice /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files; /* list of open files */
struct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



Context switch

PROCESS SCHEDULING QUEUES

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

CONTEXT SWITCH

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

S C H E D U L I N G

INTERLEAVED EXECUTION

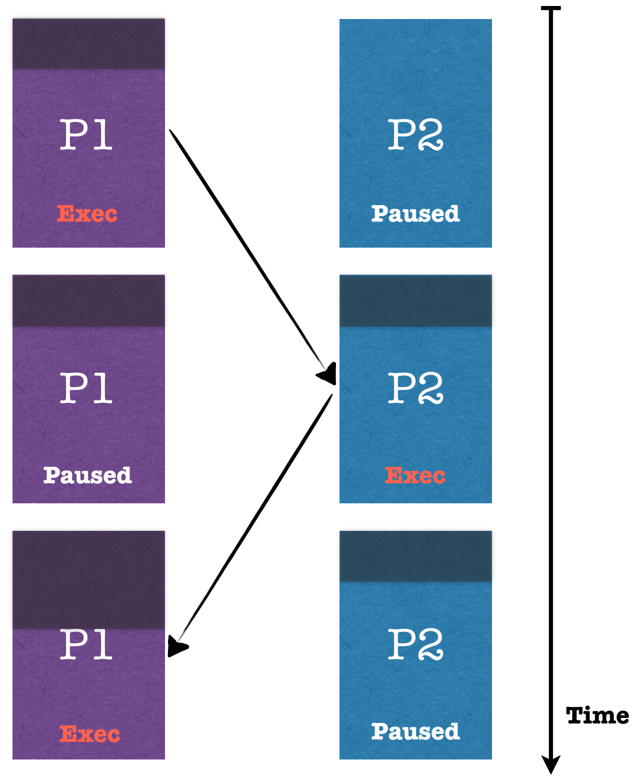
Processes take turns to be executed bit by bit

CONCURRENCY

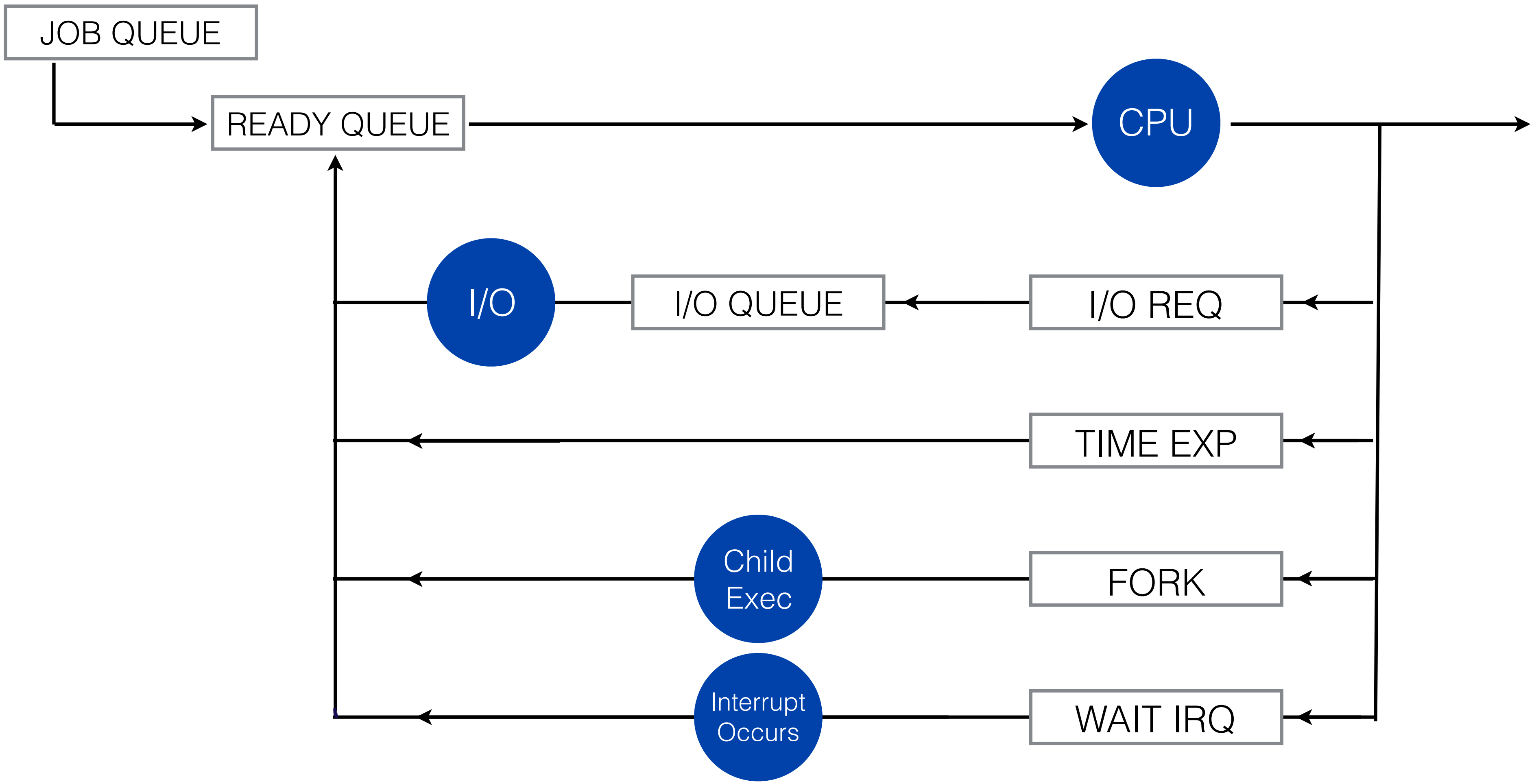
Interleaving process execution gives the illusion of concurrency, where all the programs seem to "run" at the same time

PARALLELISM

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

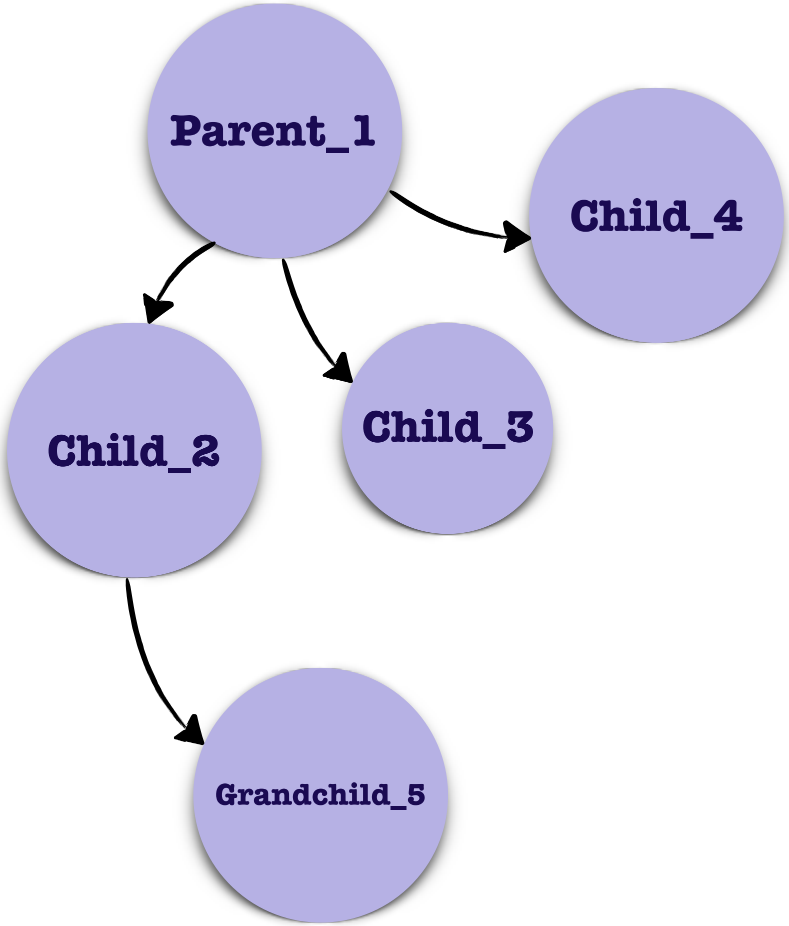


PROCESS QUEUING DIAGRAM



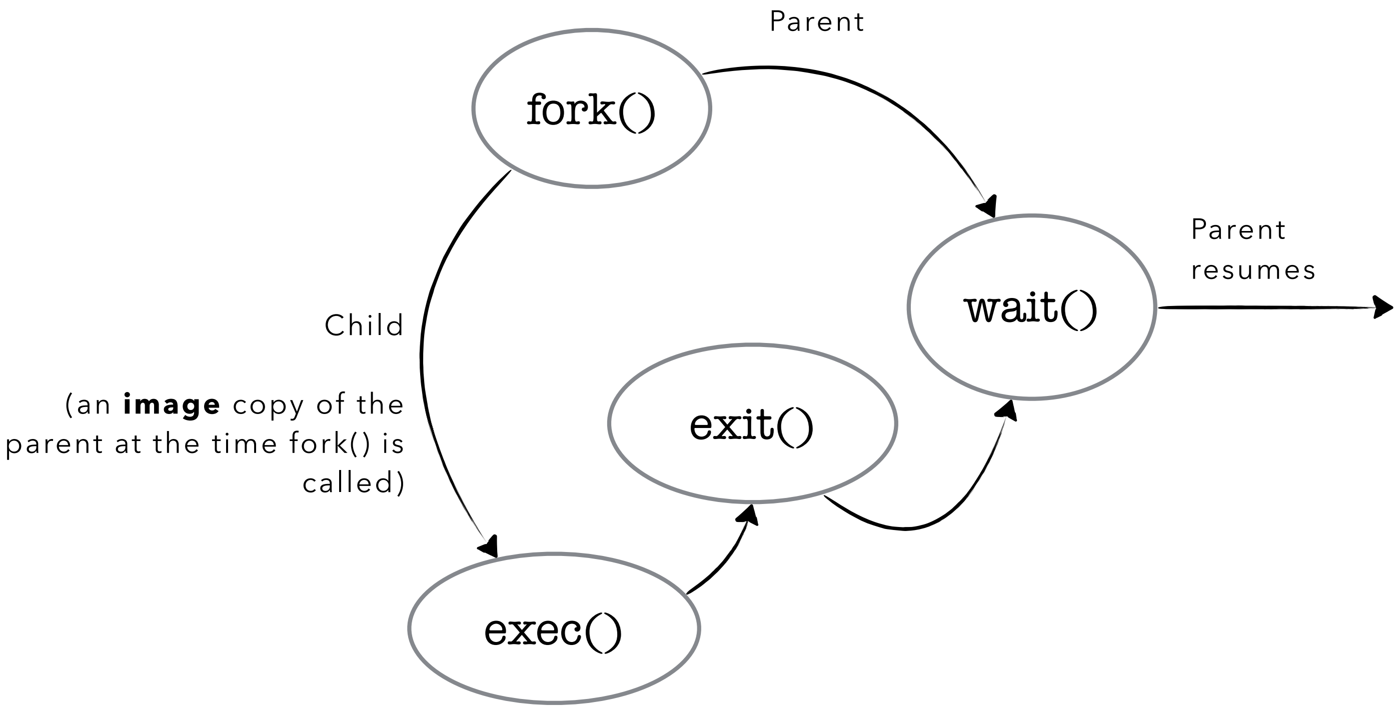
PROCESS CREATION

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

```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!

Parent

```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
}
```

Child

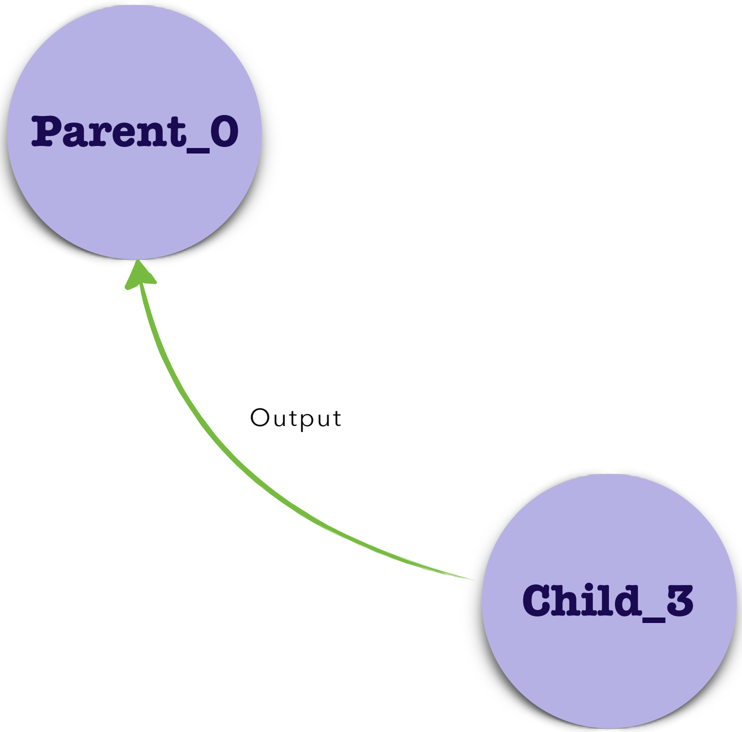
```c
void fork_example_3(){

    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

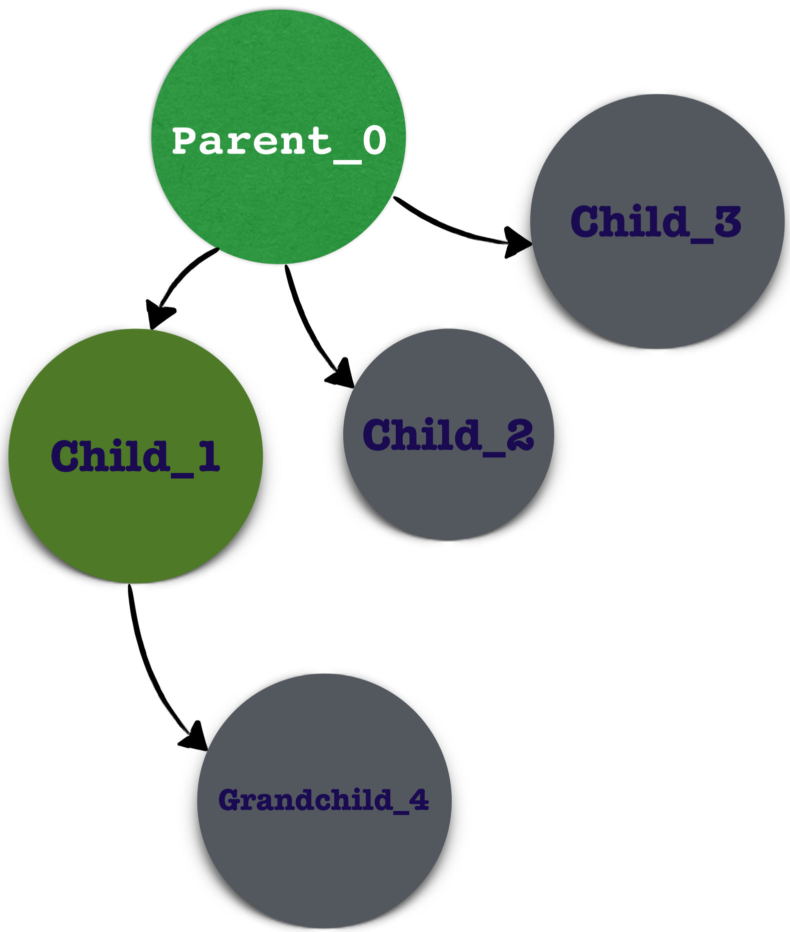
Done via **exit()** system call. Resources will be deallocated by the OS



- 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)**

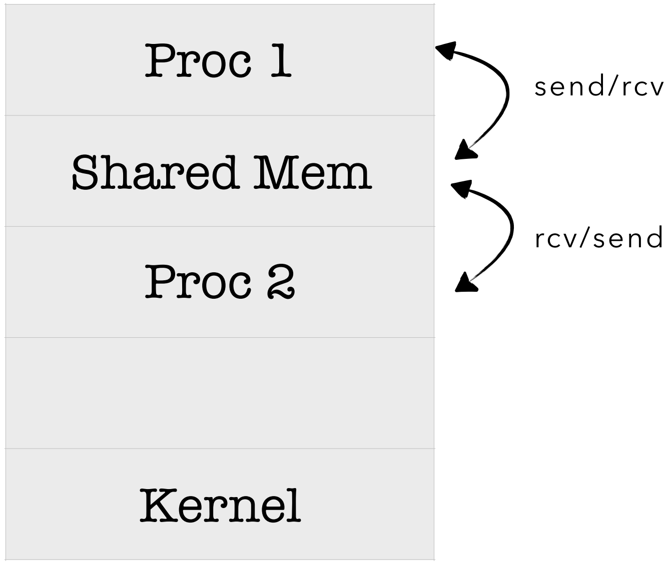
ZOMBIE PROCESSES

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

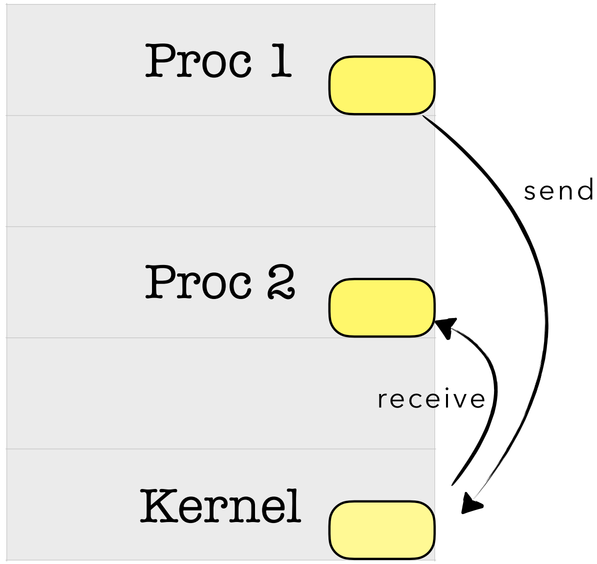


- The grey nodes: processes have terminated, but parents (in green) continue execution and does not call wait()
- All the children processes exit status is not reaped, hence turning into **zombie** processes
- OS can help abort **zombie** processes by themselves
- Otherwise, they will remain as a PCB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

INTERPROCESSES COMMUNICATION

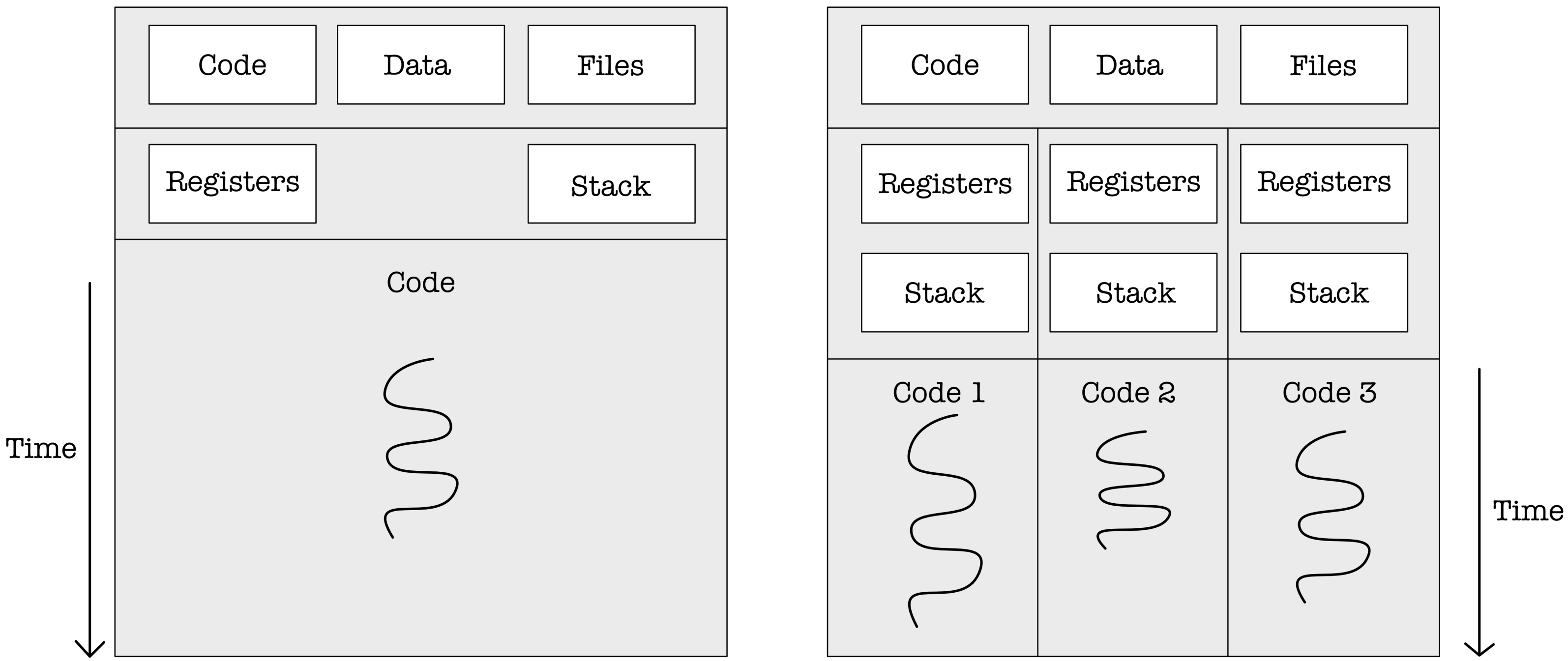


Shared memory



Sockets

THREADS

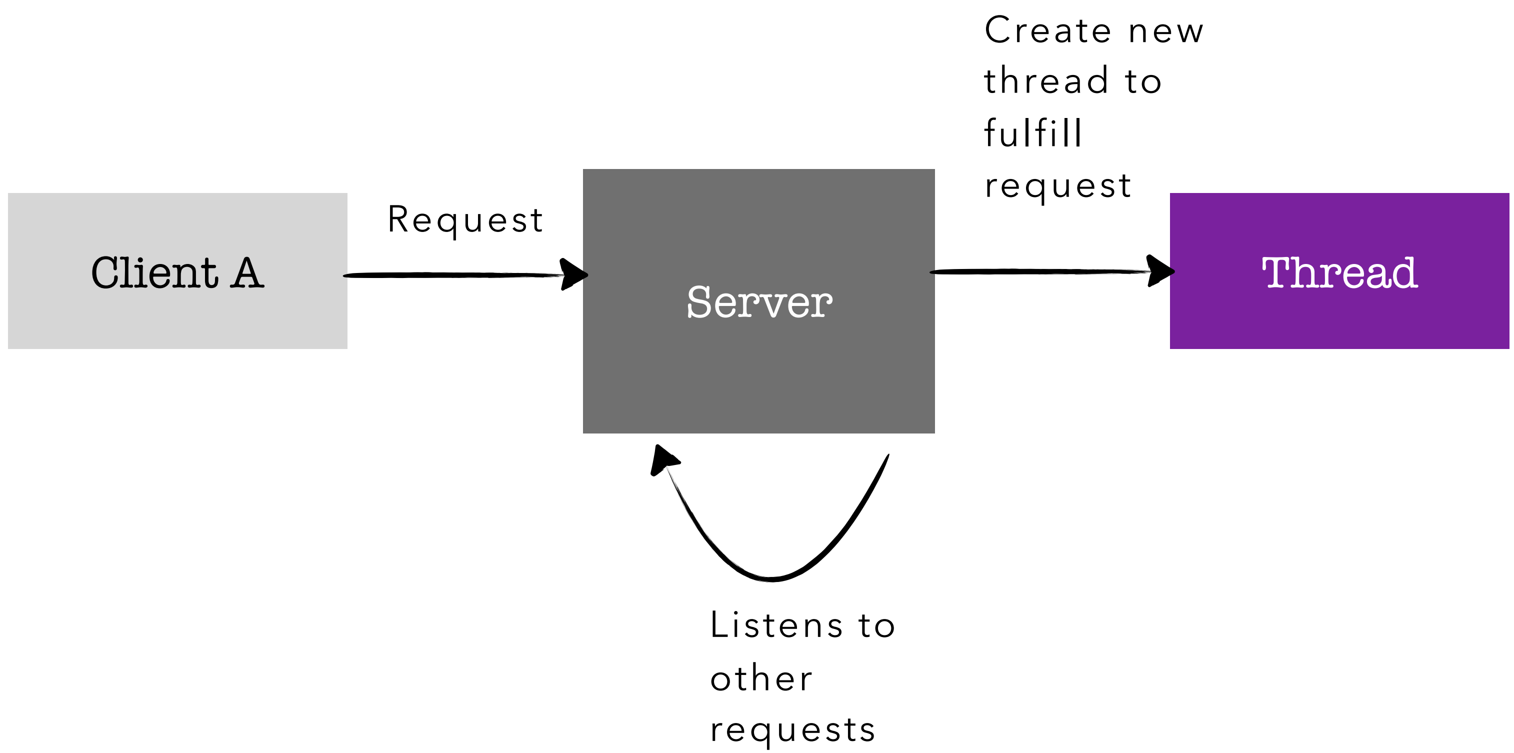


A single process with
a single thread

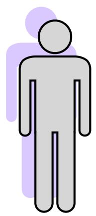
A single process with
three threads

PROCESSES	THREADS
More overhead to create	Lesser overhead to create
Requires private address space, has protection against other processes	Easier to create since they share address space, only differ in program execution
Processes are independent of one another, no concurrency issues	Requires careful programming since threads of the same process share data structures and hence are interdependent
No synchronization overhead, easier to program and work as intended	Can potentially suffer from synchronization overheads, harder to program and work as intended
Can benefit from parallel execution on multiprocessor system	Can benefit from parallel execution on multiprocessor system
A process can have many threads executing different tasks	Good for responsiveness of a process

MULTITHREADED ARCHITECTURE

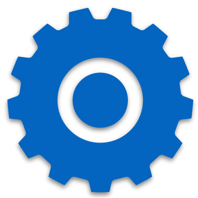


TYPES OF THREADS



User threads

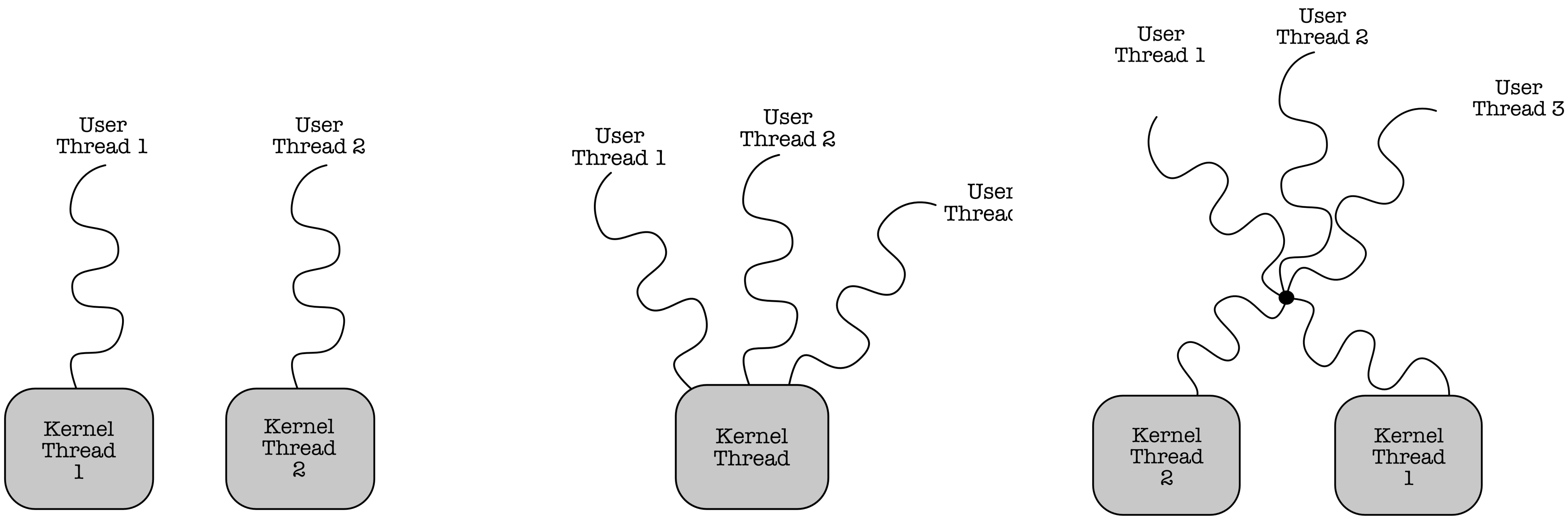
- Not known to kernel
- Scheduled by thread scheduler in thread library
- Runs in user mode, cheaper to create



Kernel threads

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

THREAD MAPPING



one to one **one to many** **many to many**