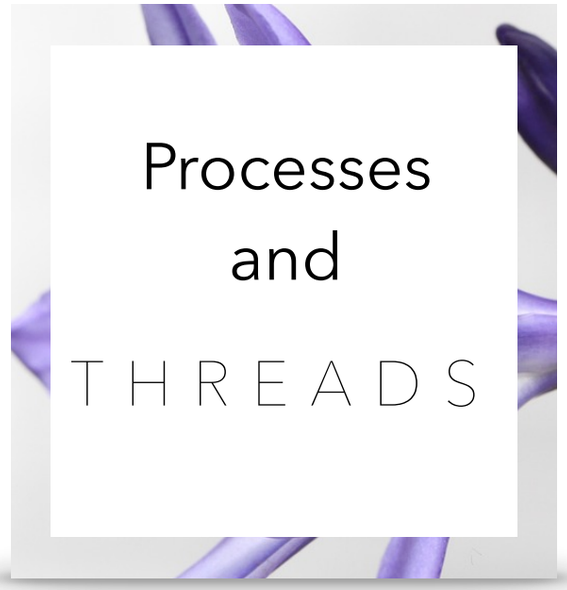


50.005 CSE

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** → The start (standard) state when you open an app + last saved state (if any)

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

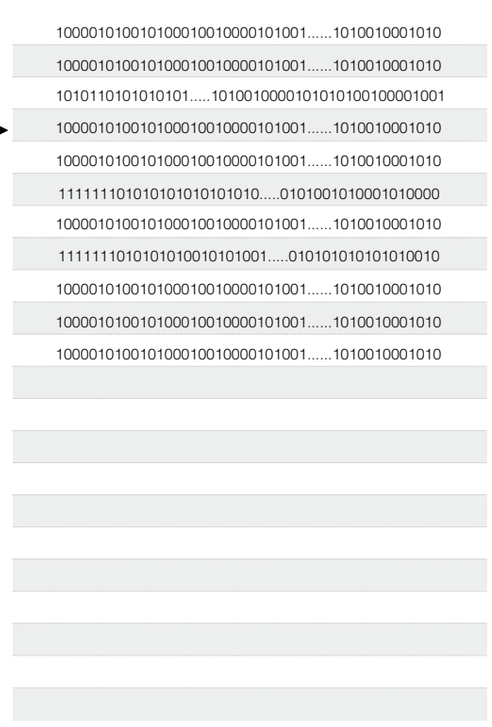
MAKING PROCESS

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

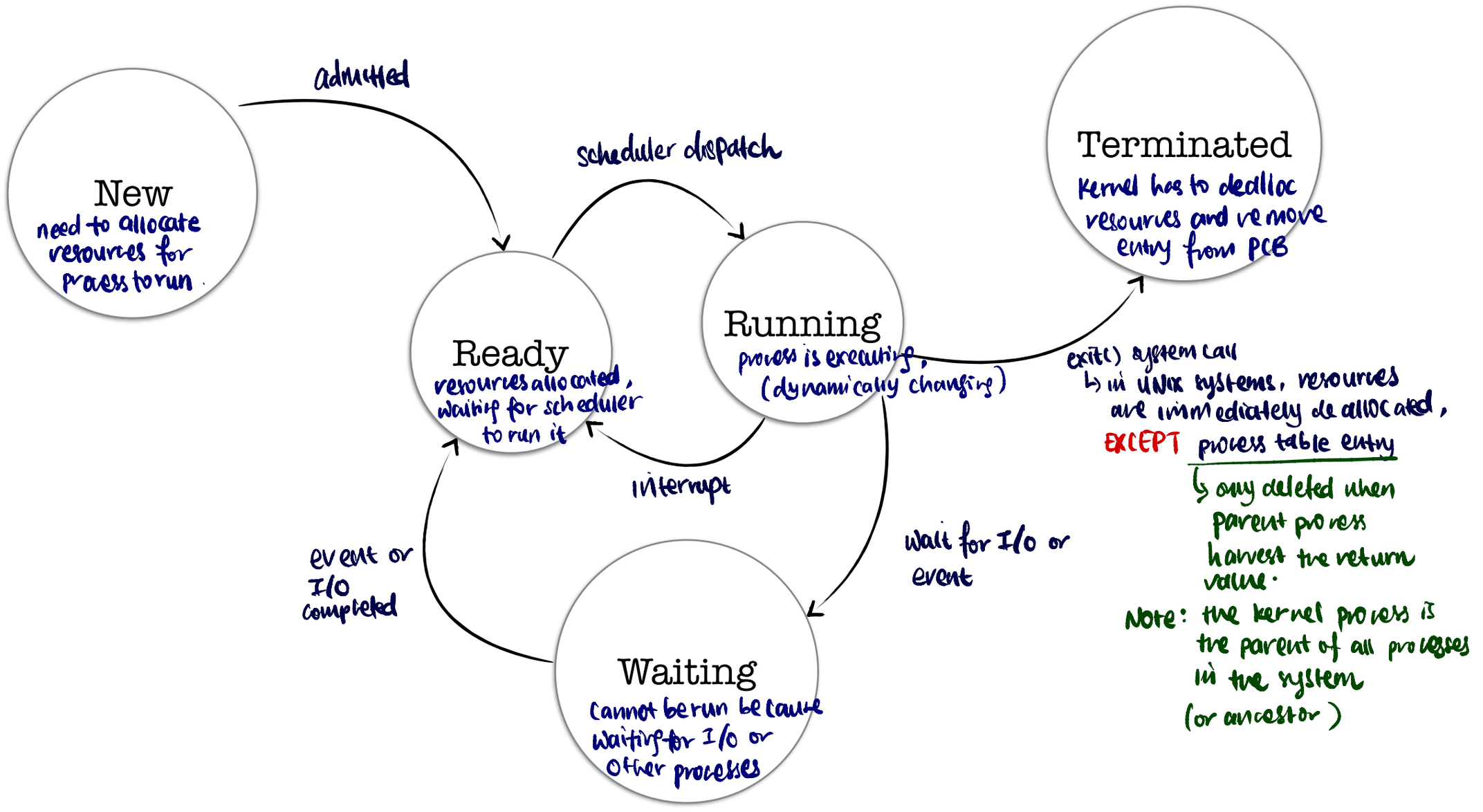
A **process image** from the executable is loaded into the **private** VM space of the process (can be on RAM or disk). This gives the initial state of a process.

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*

The name of each process in the system

new, ready, running, waiting, terminated

```
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:

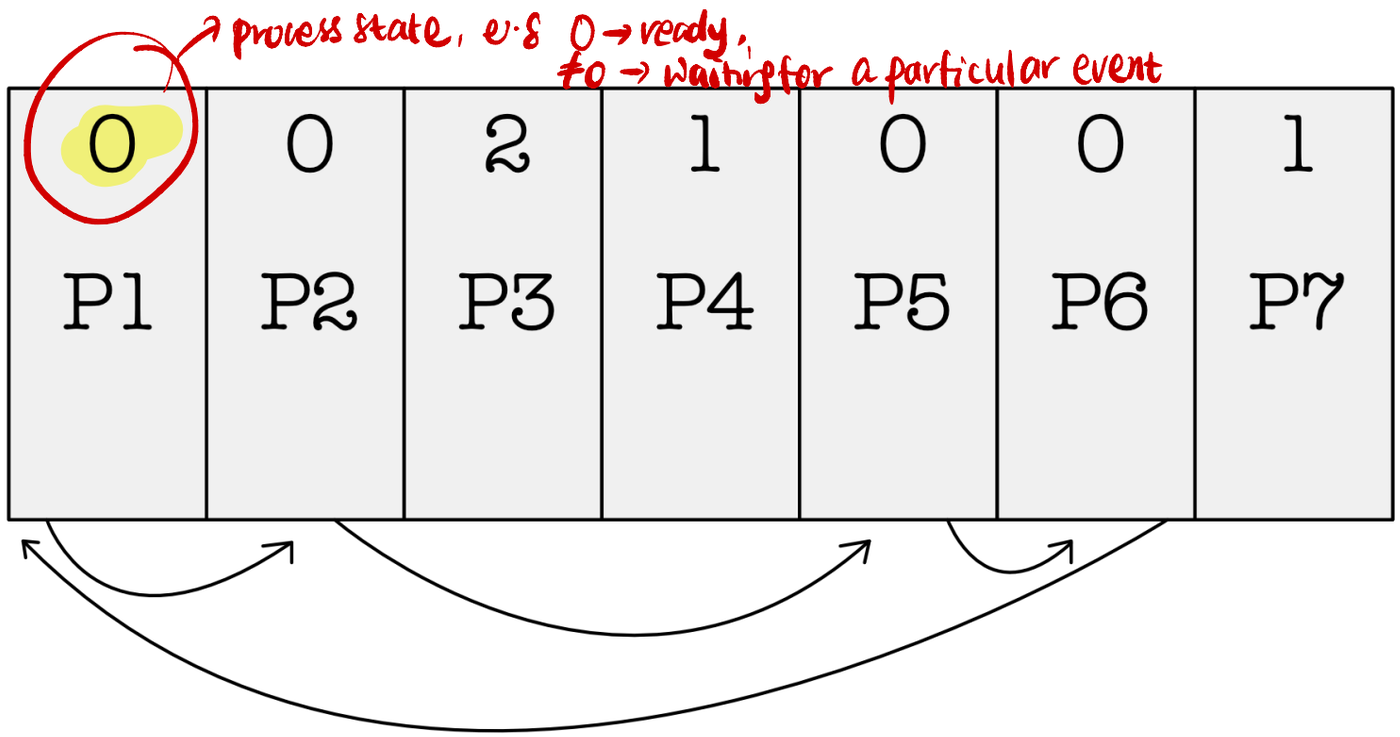
Think about why these infos are important for the scheduler and are important for context switch

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

↳ each entry in the process table is called the process control block

↳ a process table (proctable) contains the info of all processes in the system



Context switch

Step ①: save states of Pi

Step ②: load states of Pj

Step ③: run Pj

Some terminologies :

PROCESS SCHEDULING QUEUES

Three main types of queue:

the PCB contains all process info. these Queues serve as a fast lookup data structure for the scheduler to select a process.

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**
   - 1 queue per device

after selecting, will refer and update relevant PCB entries for context switch.

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

SCHEDULING

INTERLEAVED EXECUTION

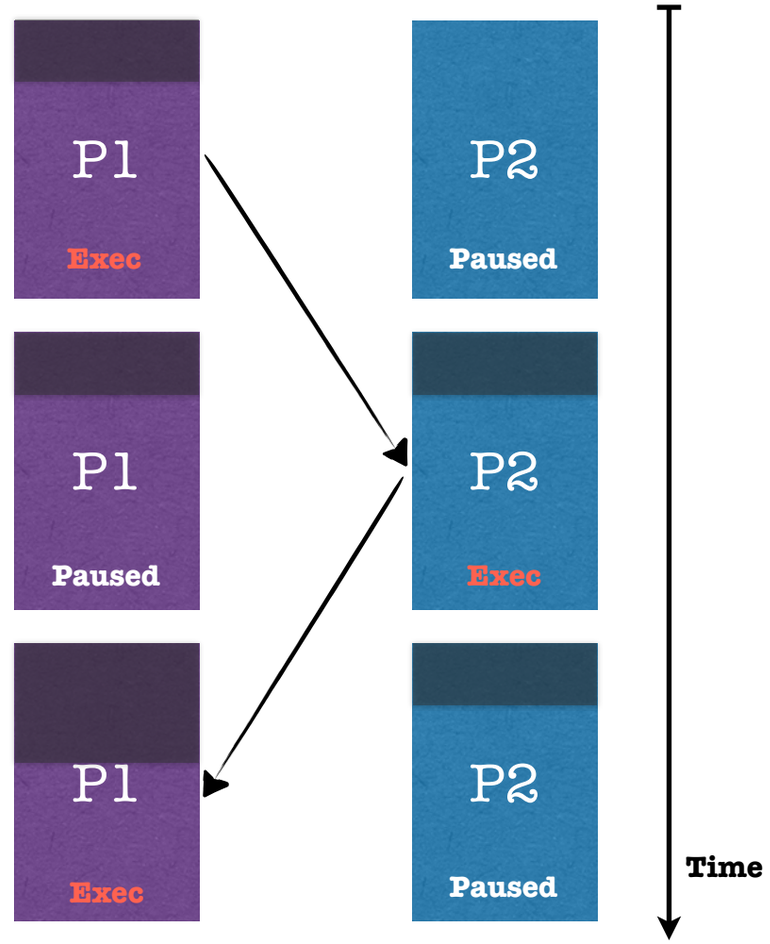
Processes take turns to be executed bit by bit

CONCURRENCY

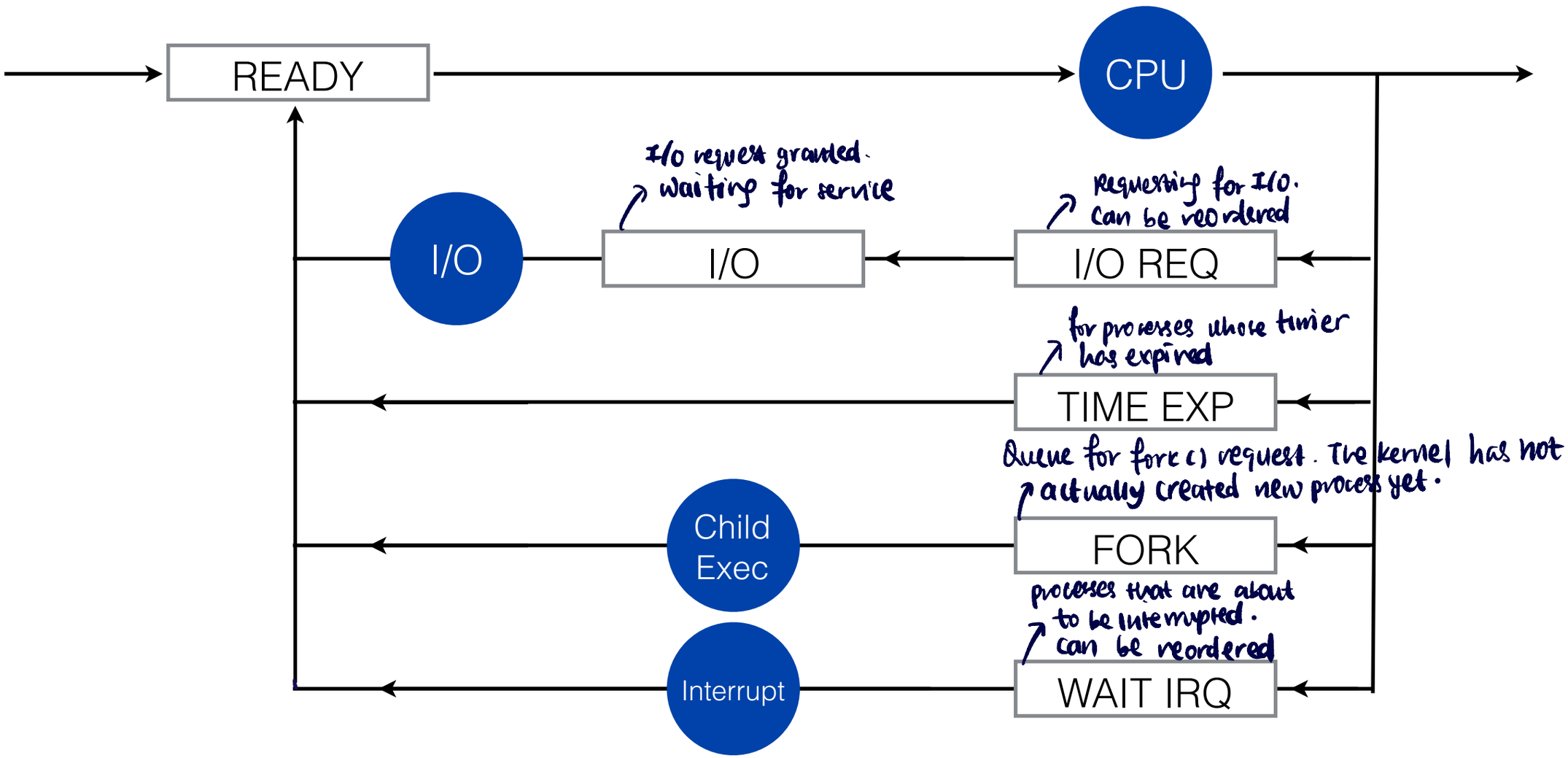
Interleaving process execution gives the illusion of concurrency, where all the programs seem to “run” at the same time

PARALLELISM

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

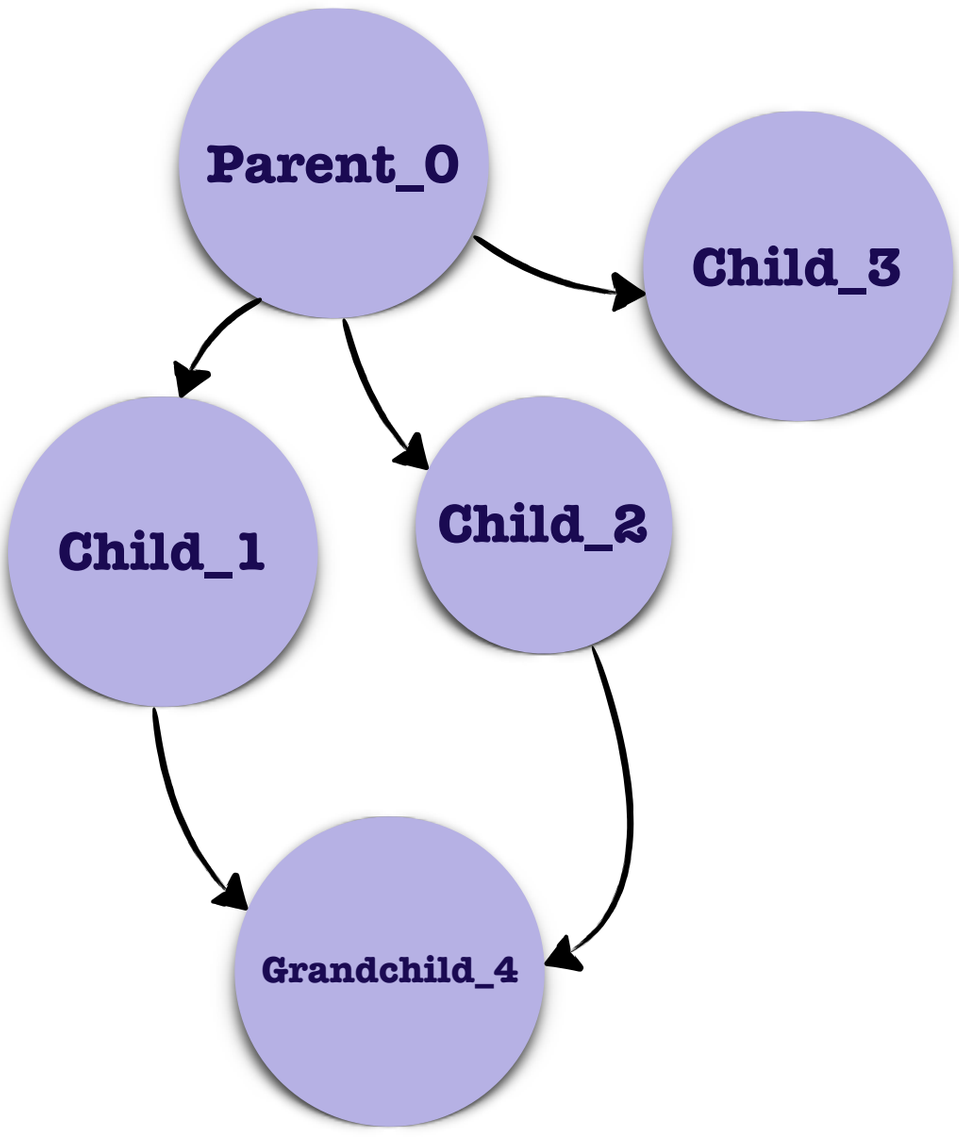


PROCESS QUEUING DIAGRAM



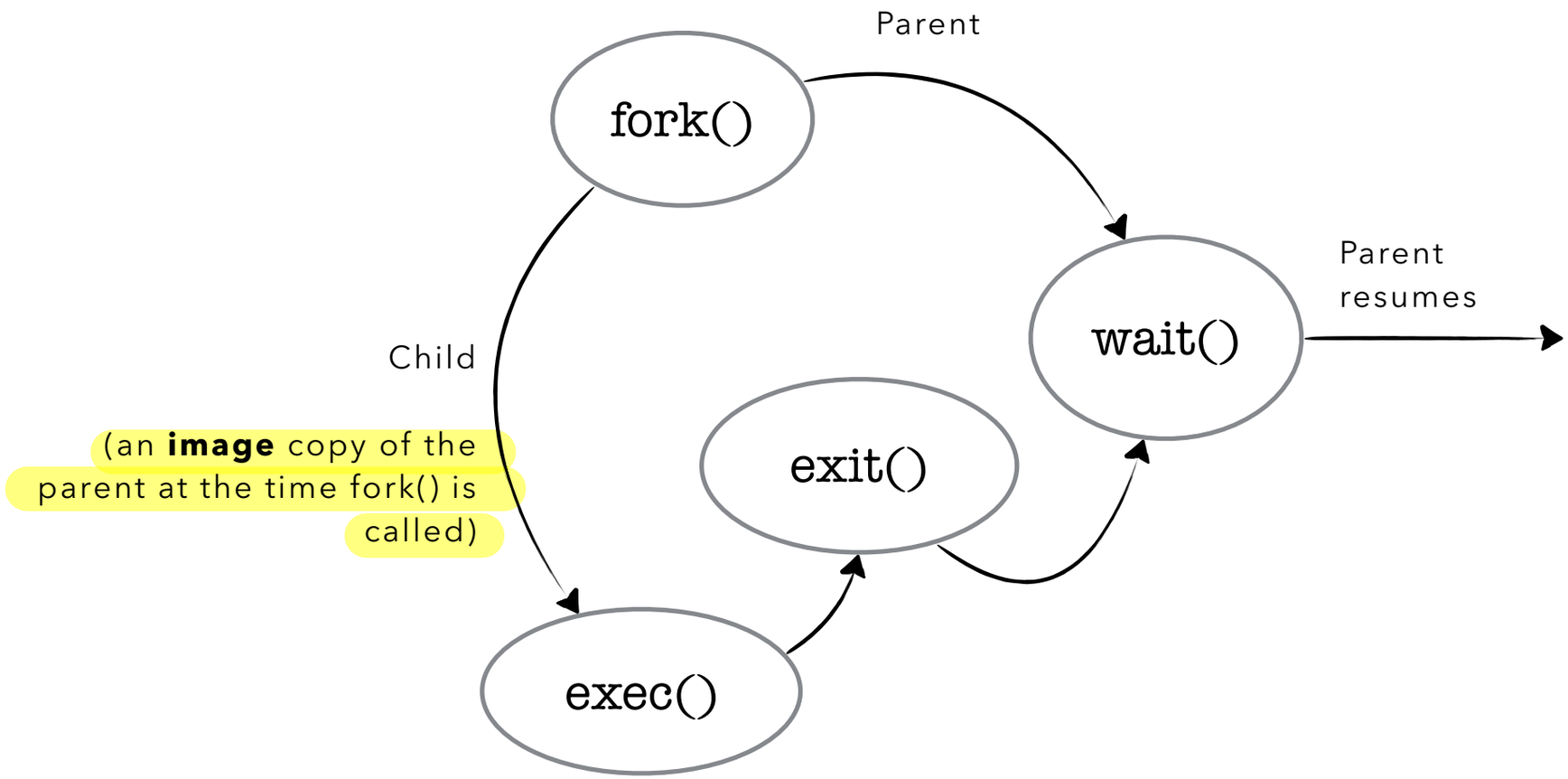
PROCESS CREATION

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

Therefore, fork() system call is necessary before exec(program) so that the new executable program does not replace the parent's process space.

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
}
```

The value of returnValue in parent process is the child's pid

Output possibility 1:

Hello from Child!
Hello from Parent!

Output possibility 2:

Hello from Parent!
Hello from Child!

There are 2 possible outputs because we do not know which process will the scheduler decide to run first.

```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
}
```

Parent

Note: see more fork() examples in the google drive

Child

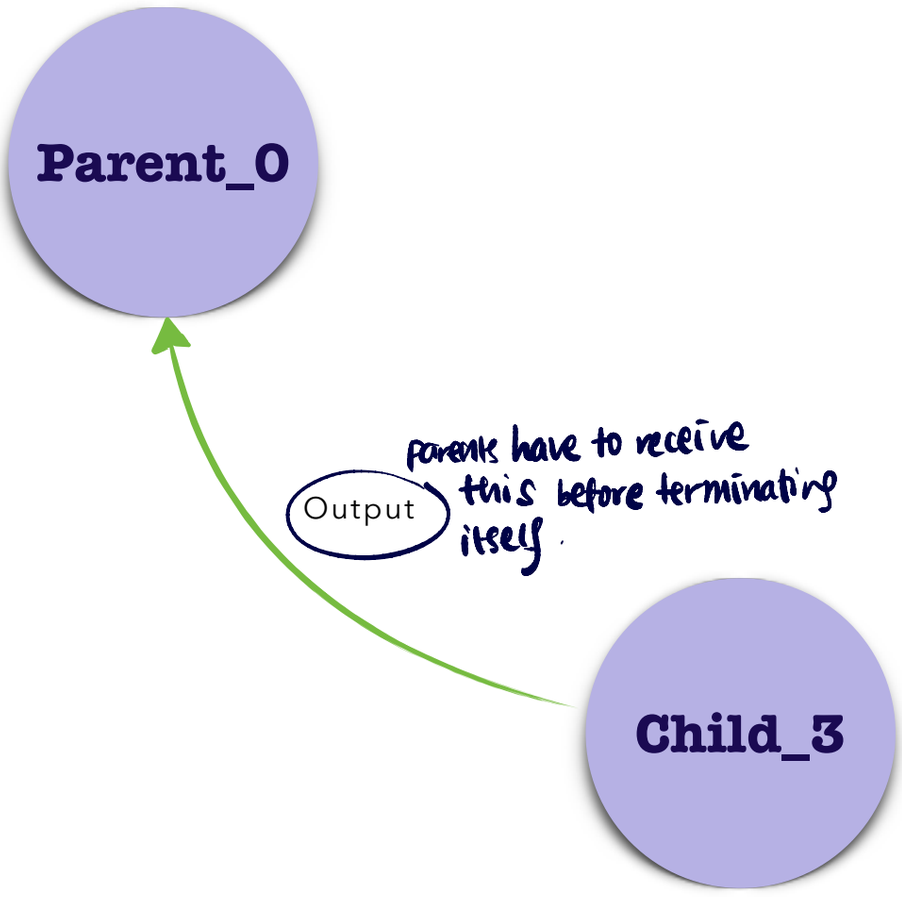
(exact copy of parent's code, continued from fork() onwards)


```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
}
```


—Entry point

PROCESS TERMINATION

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

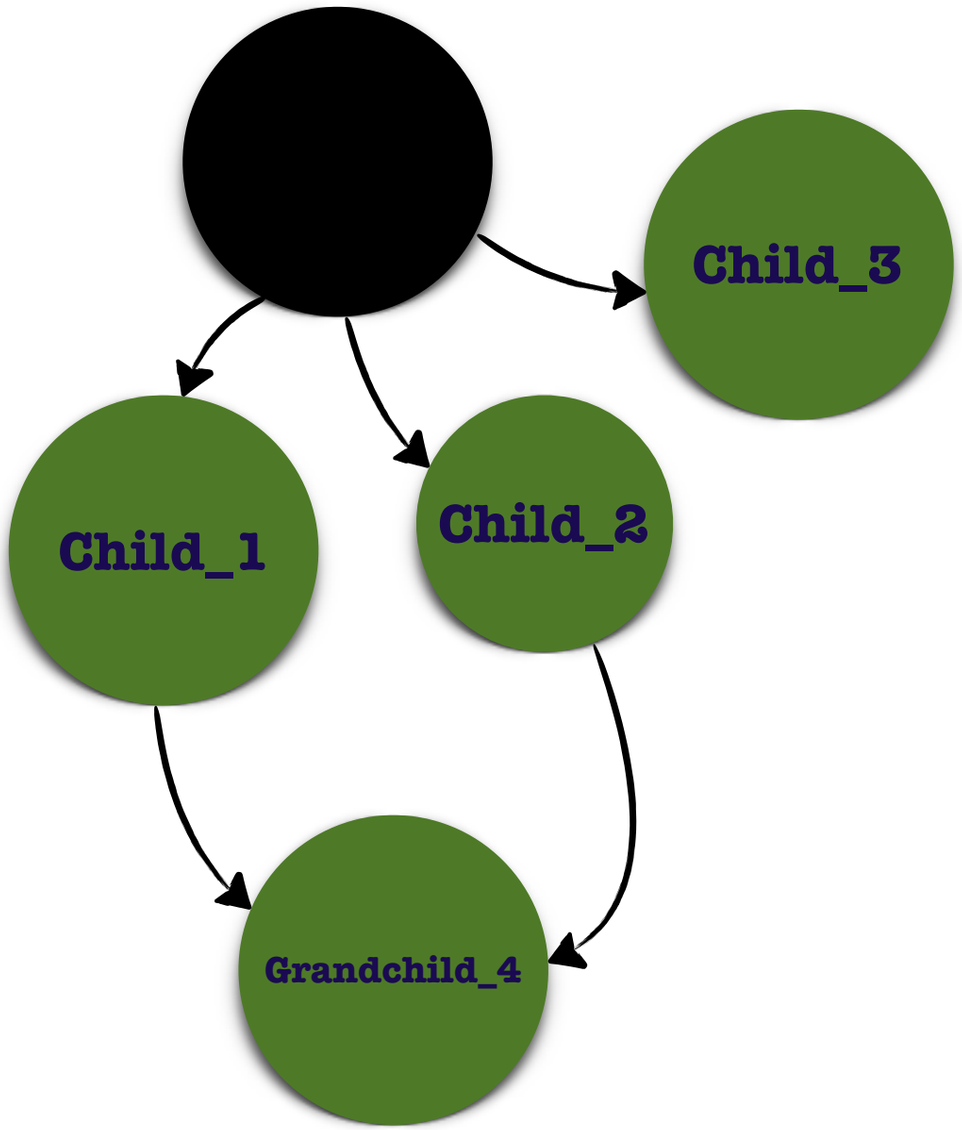


- Parent processes can abort child processes
  - If child exceeds resources (memory /time)
  - If task is no longer required
  - If parent wants to exit
- Child process can exit using system call **exit(status)**
- Usually **status** is an int indicating whether or not the exiting process is successful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

wait() & waitpid (pid) will block the caller (the parent, basically), UNTIL any child finishes (for wait()) or until the specific child (for waitpid(pid)) finishes. Otherwise if child has finished / child w that id doesn't exist / no more child exists, wait() / waitpid (pid) will return an the caller can resume normally

ZOMBIE PROCESSES

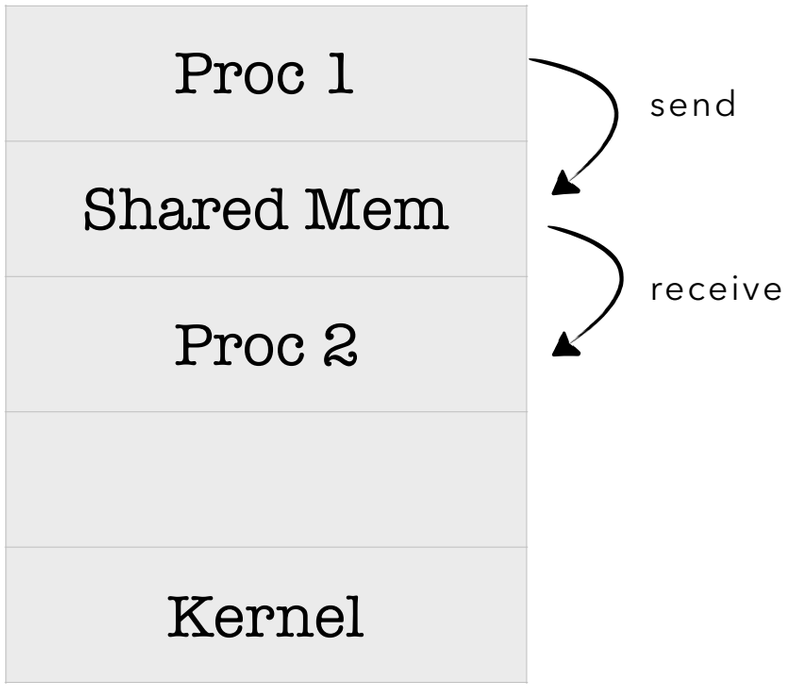
Child processes whose parents already terminated become zombie processes

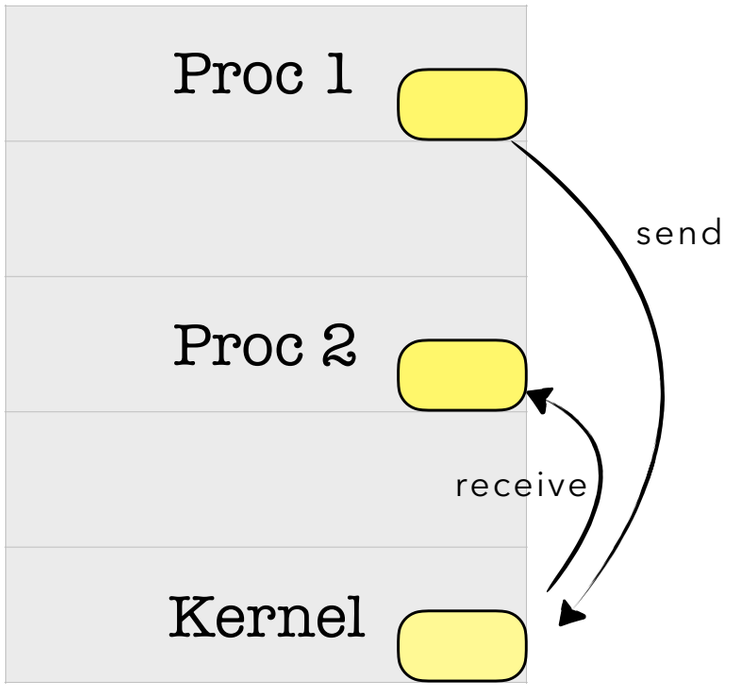


- The black node: Parent_0 has terminated
- All the children processes have nobody to return to, hence turning into **zombie** processes
  - entry of child process remains in process table although it has terminated.
- OS can help abort **zombie** processes by themselves
- Otherwise, they will take up resources until computer restarts → memory/resource leak

† so a good practice is to always wait for child process (if any) to terminate. This wait() → system call harvests the child process's exit return value and remove the child's process table entry.

INTERPROCESSES COMMUNICATION





Shared memory

1. systemcall to create shared memory region
2. Proc 1 & Proc 2 can read & write to the shared memory in user mode
3. Requires synchronization protocol so that processes do not overwrite each other

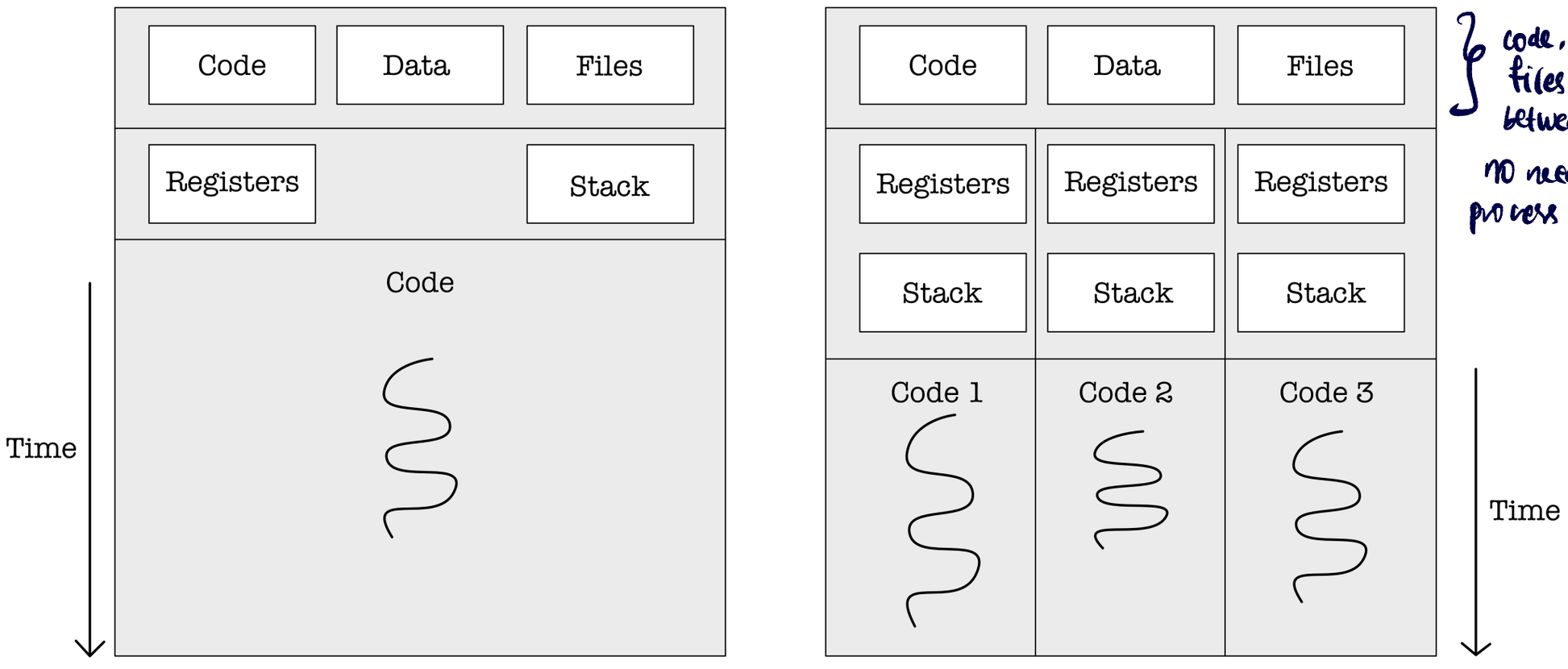
Sockets

1. Requires system call to read
2. Requires system call to write
3. Kernel will automatically synchronize process communication

Process couples (gives) concurrency and protection
- concurrency → interleaved execution
- protection → address space isolated from one another

Thread gives concurrency without protection.

THREADS



A single process with a single thread

A single process with three threads,

code, data, and files are shared between threads.

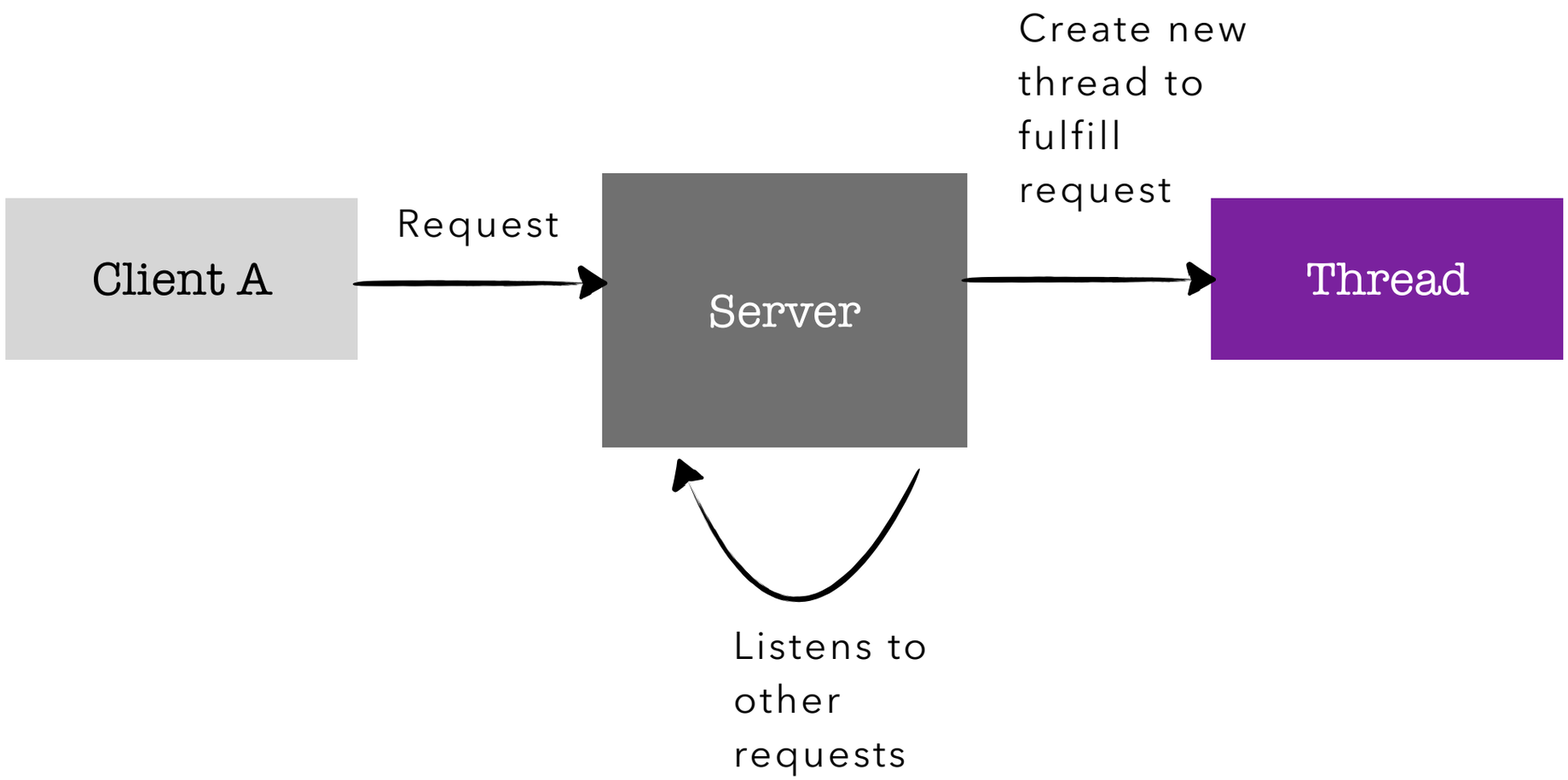
no need to copy entire process space like fork()

executing different entry points of the same code. Execution of threads of the same process is interleaved., done by the thread scheduler in user mode.

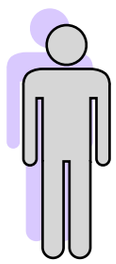
PROCESSES	THREADS
More overhead to create (because copying over the entire VM space, including code, data, & files)	Lesser overhead to create (shares data, code, and files)
Requires private address space, has protection against other processes	Easier to create since they share address space, only differ 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

one of the uses of threads



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

Way larger than user thread data structure

perform kernel thread's own

* runs in kernel mode

Note: if one kernel thread is blocked, the other kernel threads that are runnable can run.
User mode cannot interrupt kernel mode.
However within kernel mode itself, the kernel scheduler is free to choose which kernel thread to run / pause / kill.

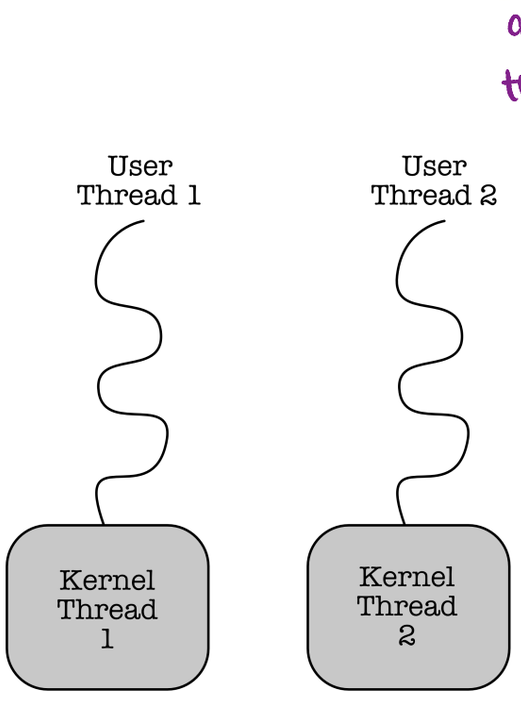
Note: multiple kernel threads are required to achieve parallelism if the system has multiple cores.

↓

because a single kernel process oversees ALL hardwares, so to use different cores the kernel process has to create kernel threads.

THREAD MAPPING

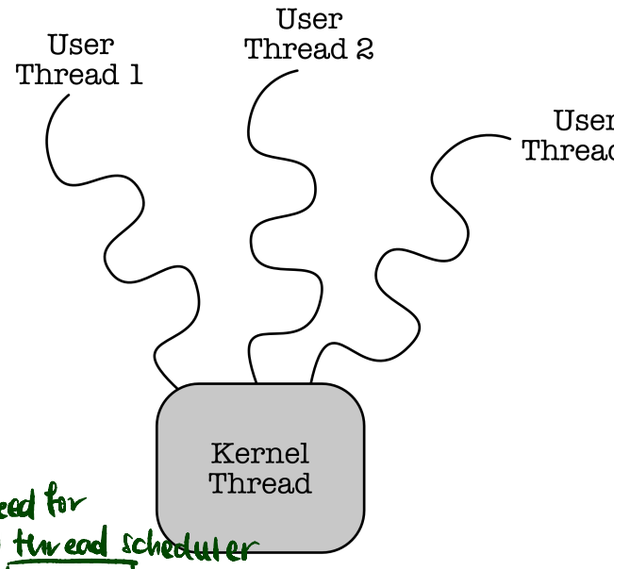
Thread mapping is SUPPORTED by the programming language's API (thread library). if you want your app to make use of your multicores then choose a library that allows either one to one mapping or many to many mapping.



one to one

✓ maps each user thread to kernel thread
✓ provides more concurrency than many to one model
✓ allows multiple threads to run on multiprocessor

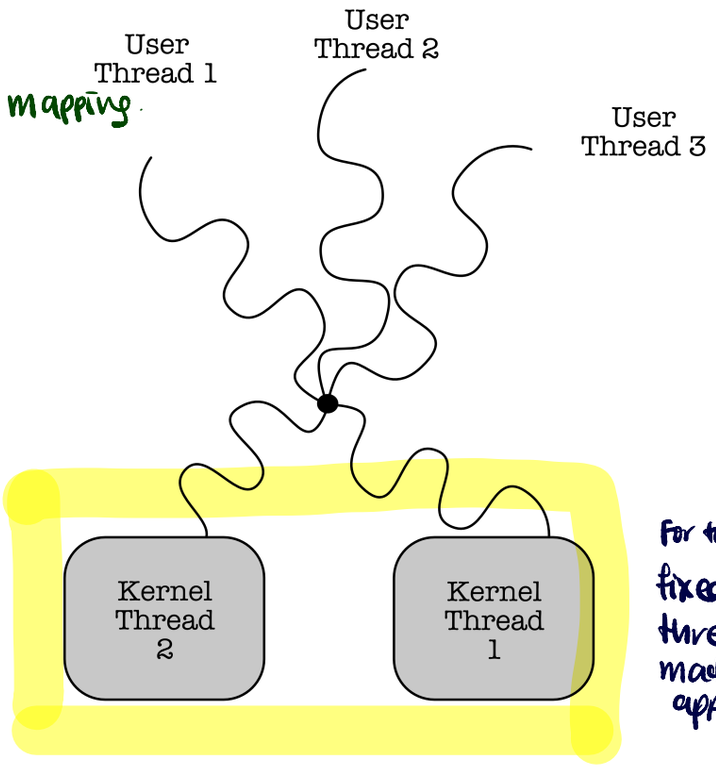
cons:
✗ kernel thread creation overhead. Developer may not be able to create too many user threads. (Kernel threads are large in size)



pros: no need for kernel thread scheduler cheap, but only 1 thread can access the kernel at a time. multiple threads aren't able to run on multiprocessors

one to many

cons:
✗ an entire process will block if a thread in that process makes a blocking system call.

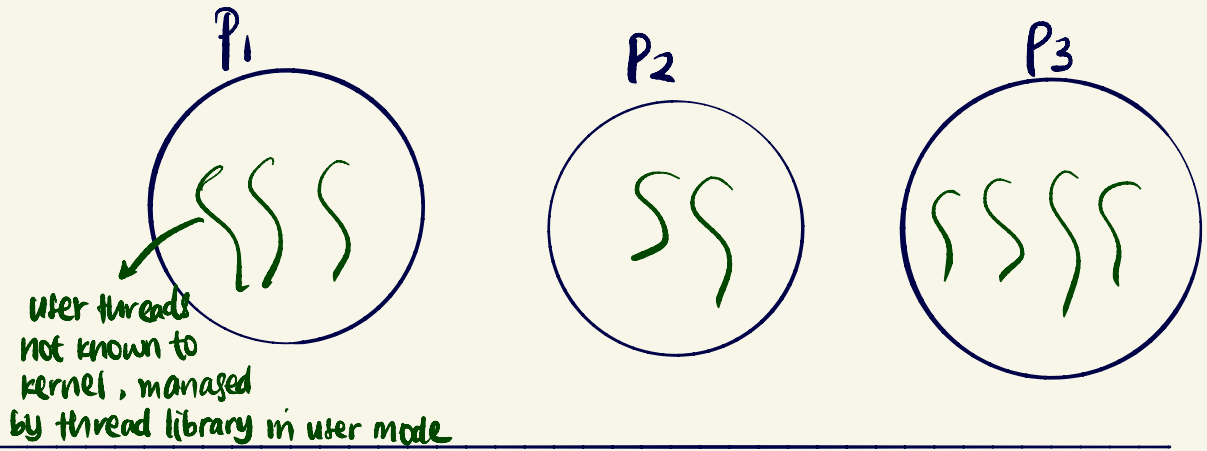


For this model, fixed kernel thread per machine or per app.

many to many (BEST)

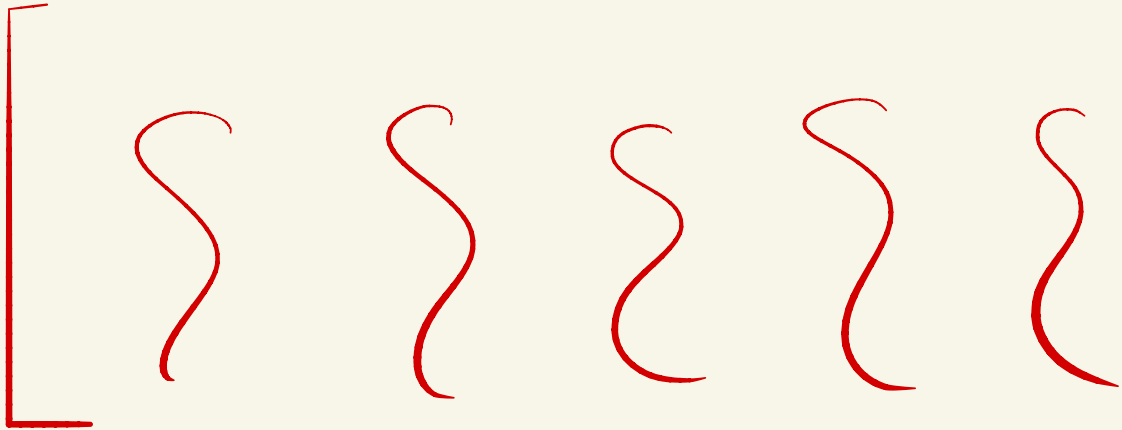
user can create as many user threads, but true parallelism (true concurrency) isn't gained because the kernel can only schedule a process → then a user thread at a time.

User mode
(in user space)

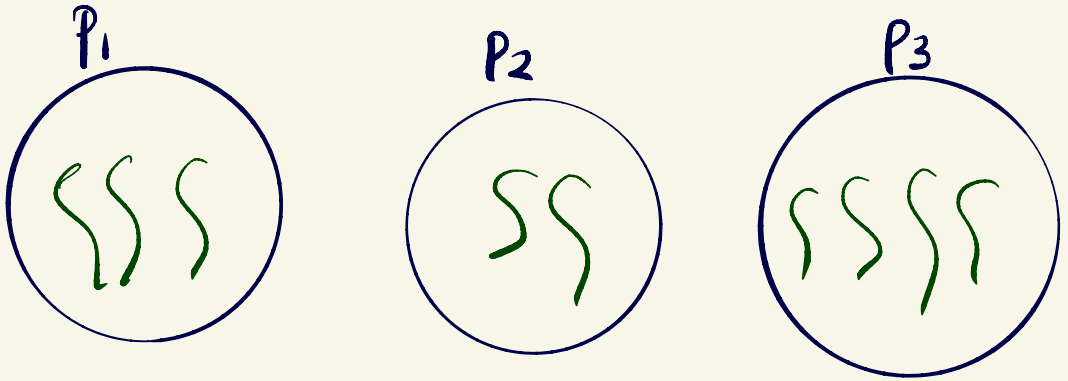


Kernel mode
(in Kernel space)

Kernel only has a SINGLE process but can have multiple kernel threads to exec diff tasks / access diff cores

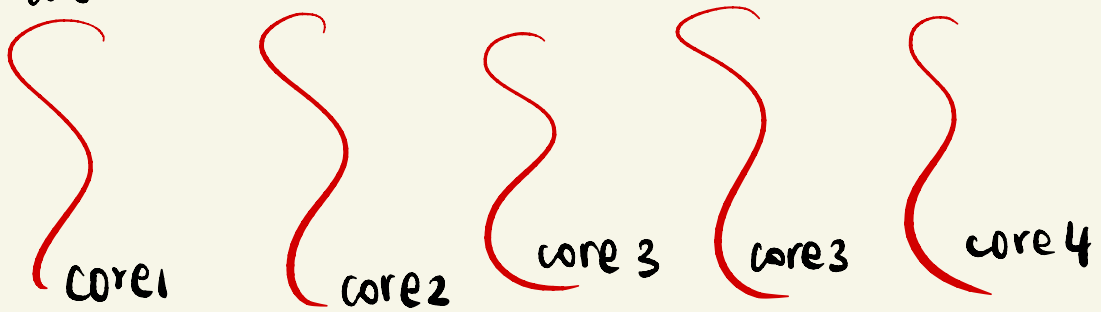


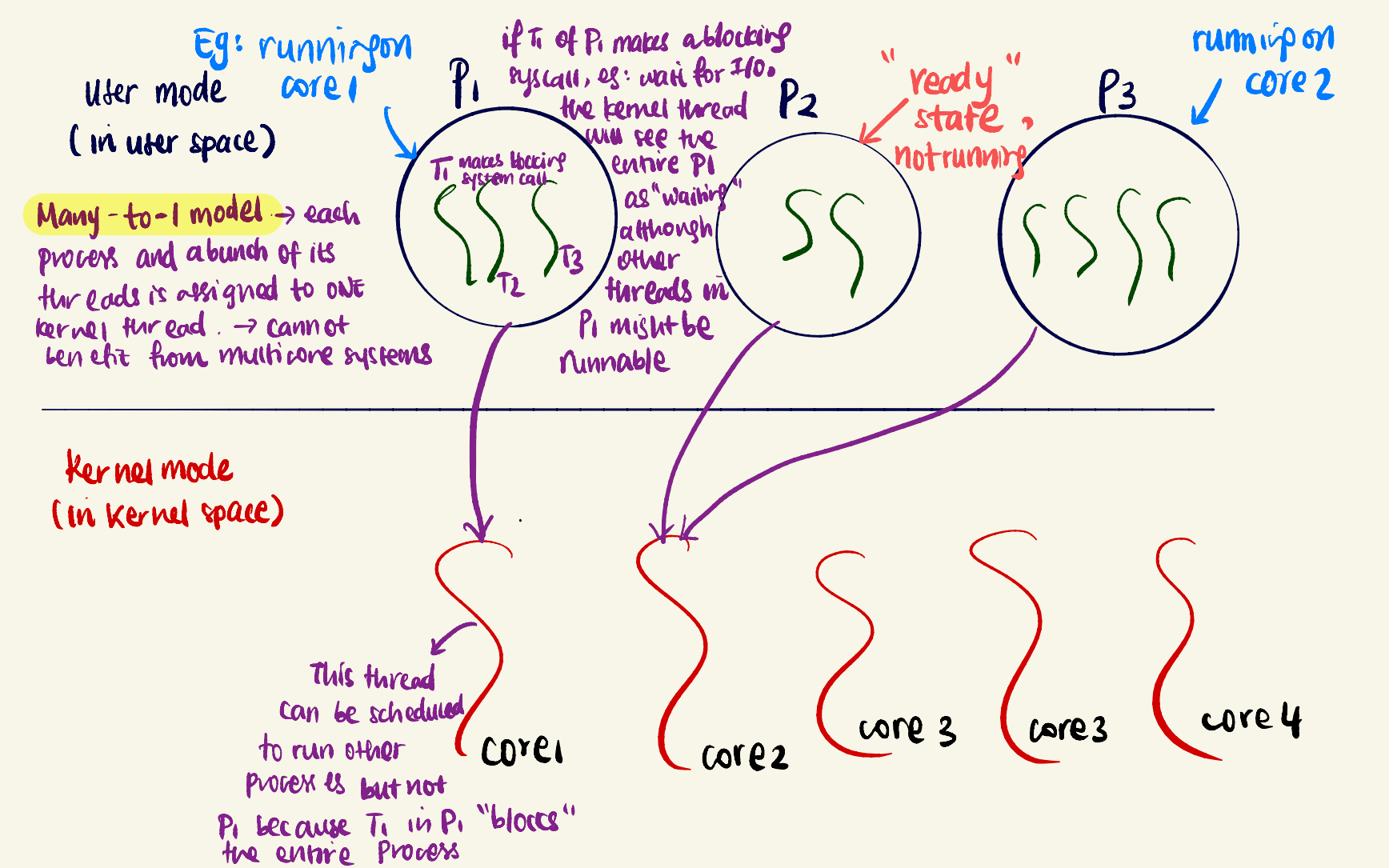
User mode
(in user space)

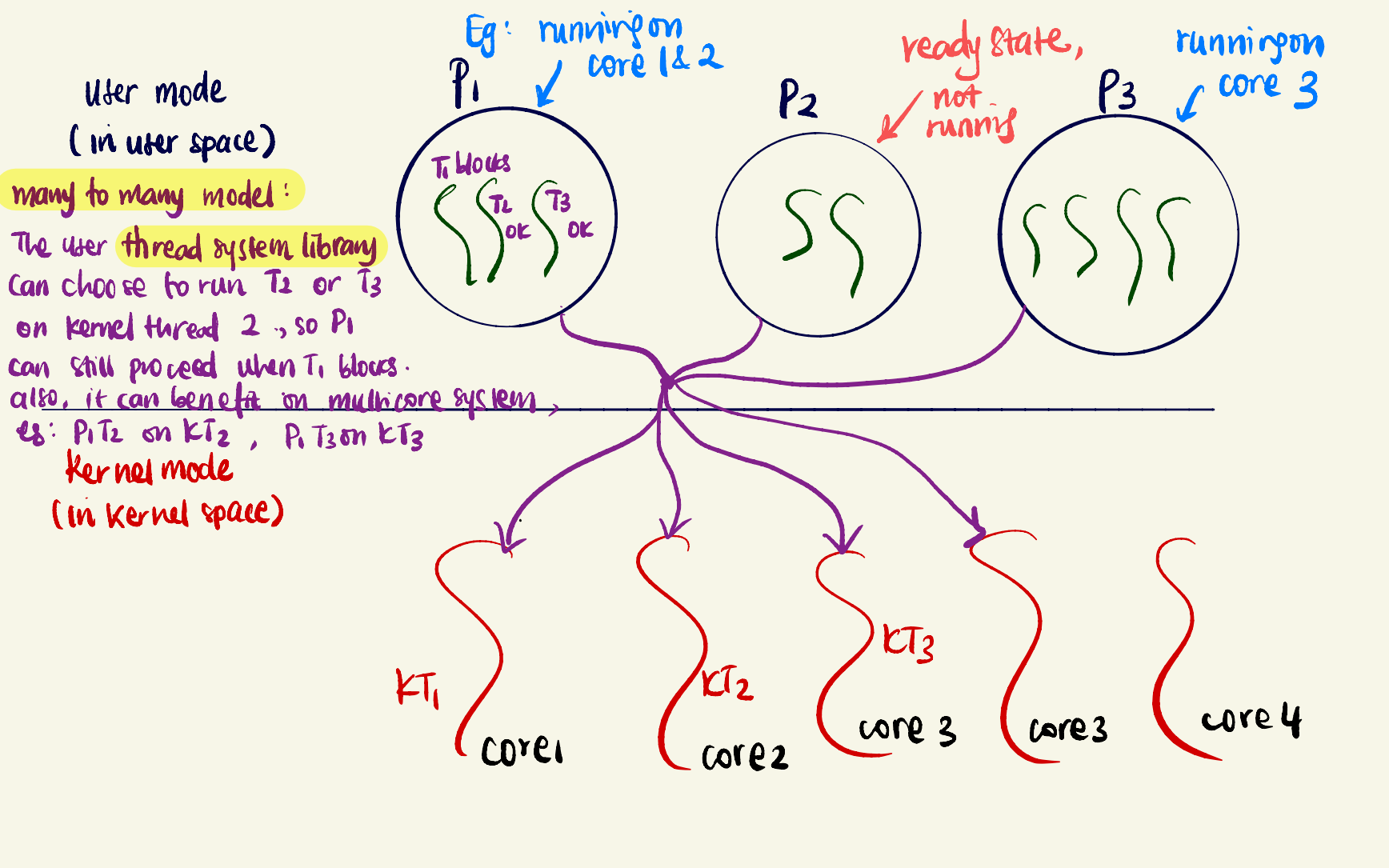


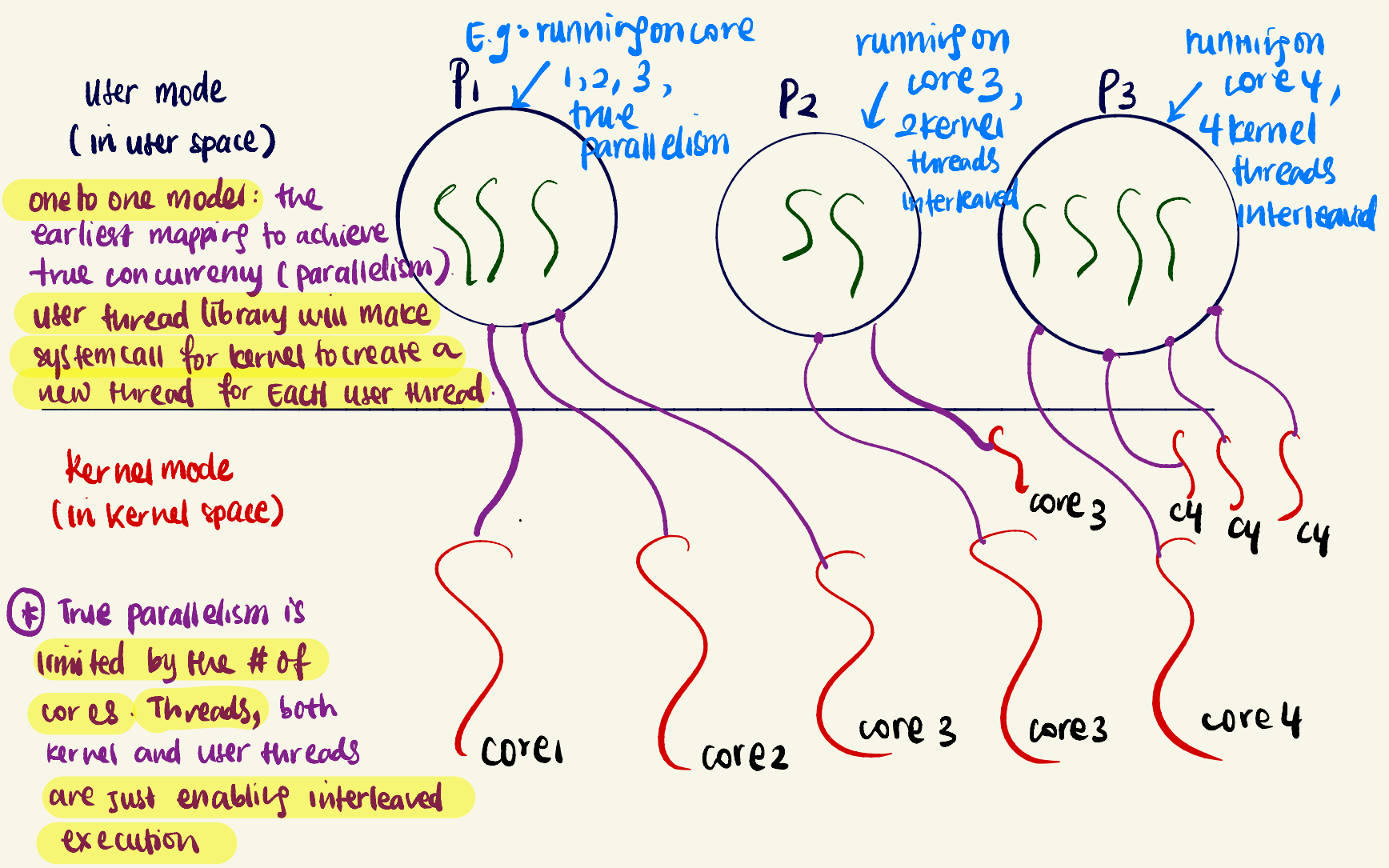
Kernel mode
(in Kernel space)

The kernel can benefit from multicore hardware iff it can create at least 1 kernel thread assigned per core.




Eg: running on core 1
User mode
(in user space)
Many-to-1 model → each process and a bunch of its threads is assigned to ONE kernel thread. → cannot benefit from multicore systems
P1
T1 makes blocking system call
T2
T3
if T1 of P1 makes a blocking syscall, eg: wait for I/O, the kernel thread will see the entire P1 as "waiting" although other threads in P1 might be runnable
P2
"ready" state, not running
P3
running on core 2
Kernel mode
(in kernel space)
This thread can be scheduled to run other process but not P1 because T1 in P1 "blocks" the entire process
core1
core2
core 3
core3
core4

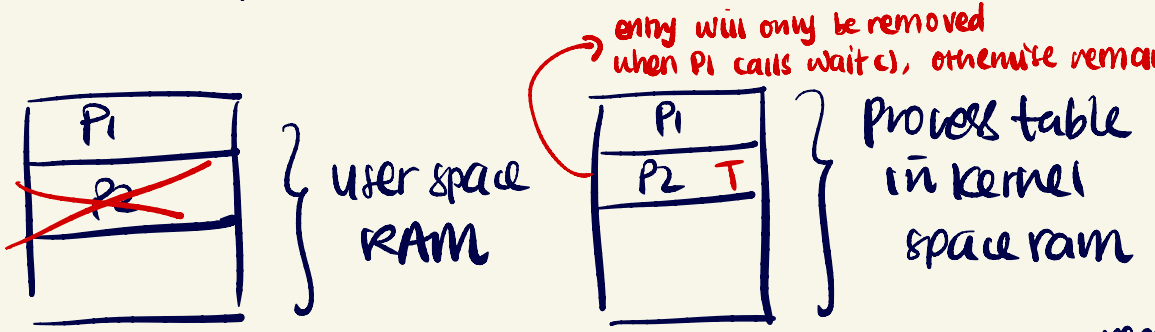

User mode
(in user space)
many to many model:
The user thread system library
can choose to run T2 or T3
on kernel thread 2, so P1
can still proceed when T1 blocks.
also, it can benefit on multicore system
eg: P1T2 on KT2, P1T3 on KT3
Kernel mode
(in kernel space)
Eg: running on core 1&2
P1
T1 blocks
T2 OK
T3 OK
P2
ready state, not running
P3
running on core 3
KT1
KT2
KT3
core1
core2
core 3
core3
core4


User mode
(in user space)
one to one model: the earliest mapping to achieve true concurrency (parallelism)
user thread library will make system call for kernel to create a new thread for EACH user thread
Kernel mode
(in Kernel space)
E.g. running on core 1, 2, 3, true parallelism
P1
P2
running on core 3, 2 kernel threads interleaved
P3
running on core 4, 4 kernel threads interleaved
core1
core2
core 3
core3
core3
core4
c4
c4
c4
* True parallelism is limited by the # of cores. Threads, both kernel and user threads are just enabling interleaved execution

Summary :

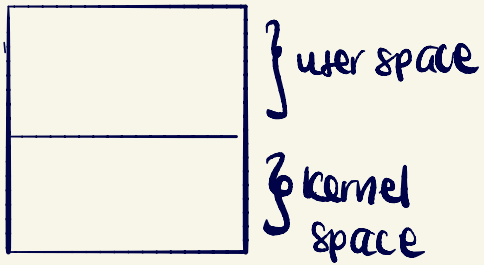
Parent process has to wait() for child process

code → (compiled) → Program → (exec) → process → (fork()) → child process



If P1 —fork()— P2, when P2 calls exit(), P2's user space will be deallocated, P2 will be labelled as Terminated (T), but entry in proctable remains

Note: recall, RAM is divided into user and kernel space.



* The process table contains process control blocks (has pid, process states) for kernel to schedule processes.
* There are 3 queues: job, device, ready queue
  - to quickly determine which process is next, then scheduler refers to the corresponding block on process table for further info.

* Process (scheduled by kernel) provides:
  1. concurrency
  2. isolation / protection (expensive) → need socket / shared mem for IPC
* Thread provides: concurrency without protection (shares code, data, and files) → will need to implement extra measures to synchronize when coding apps that creates multiple threads within a single process.
  - scheduled by thread library in user mode. → responsible to implement whichever thread mapping suitable for the user