Operating Systems
CS 415

Lecture 5:
Process Scheduling

Suyash Gupta
Assistant Professor
Distopia Labs and ONRG
Dept. of Computer Science
(E) suyash@uoregon.edu
(W) gupta-suyash.github.io

Announcements

- **Suyash Gupta**
  - Office Hours: Thursday, 10-11am, Deschutes 334

- **Nihal Balivada (TA)**
  - Office Hours: Wednesday/ Friday, 11-12pm, Deschutes 335

- **Ranjitha Rani (TA)**
  - Office Hours: Monday /Tuesday, 1-2pm, Deschutes 229

Assignment 1 is Out!

- **Deadline → April 21, 2026 at 11:59pm PST**
- Please start working and talk to us if you are stuck.

- **Midterm → April 28, 2026 (in-class)**
  - Closed book, no cheat sheets, no discussions.

Last Class

- Process

- Process Structures and States

Why do we need a Process Scheduler?

Process Scheduler

- Selects an available process for execution on a core.
    - Possibly from a set of several available processes.
    - Each CPU core can run one process at a time.

- Examples: In multiprogramming and timesharing.

Degree of Multiprogramming

Degree of Multiprogramming

- The number of processes currently in memory.
- A multicore system can run multiple processes at one time.
- If there are more processes than cores, excess processes will have to wait until a core is free and can be rescheduled.

Process Types

Process Types

- Processes can be either I/O bound or CPU bound.

- **I/O-bound process** → spends more of its time doing I/O than computations.

- **CPU-bound process** → generates I/O requests infrequently, using more of its time doing computations.

Types of Schedulers

Types of Schedulers

- There are three levels of scheduling:

- **Long-term scheduling →** decision about which processes are admitted into system.
  - Usually employed in batch systems.

- **Medium-term scheduling →** decision about which processes are memory resident.
  - May temporarily remove processes from memory and later return them to memory.

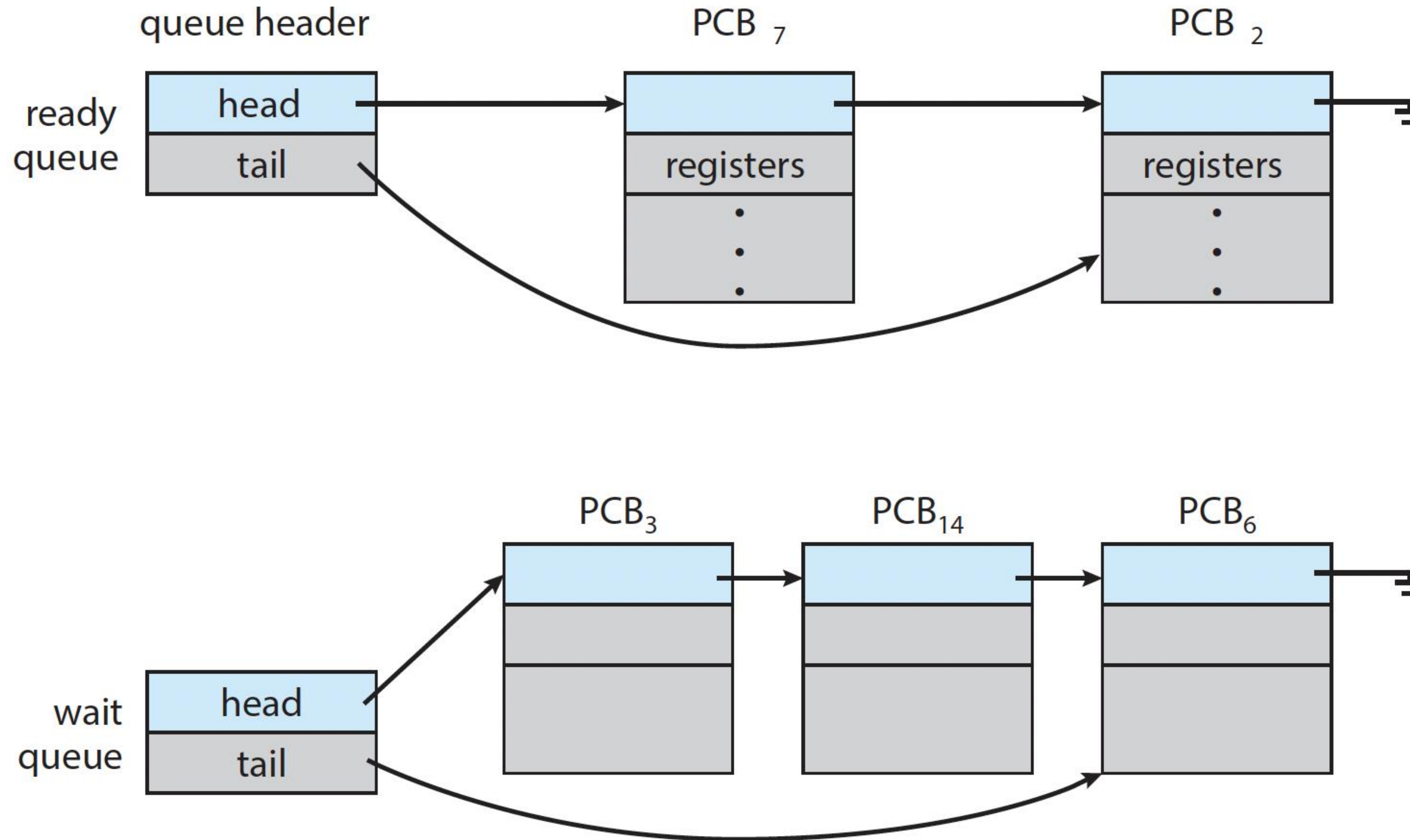
- **Short-term scheduling →** decision about which memory resident process gets the CPU next.
  - Short-term scheduler is also called process scheduler

Process Queues

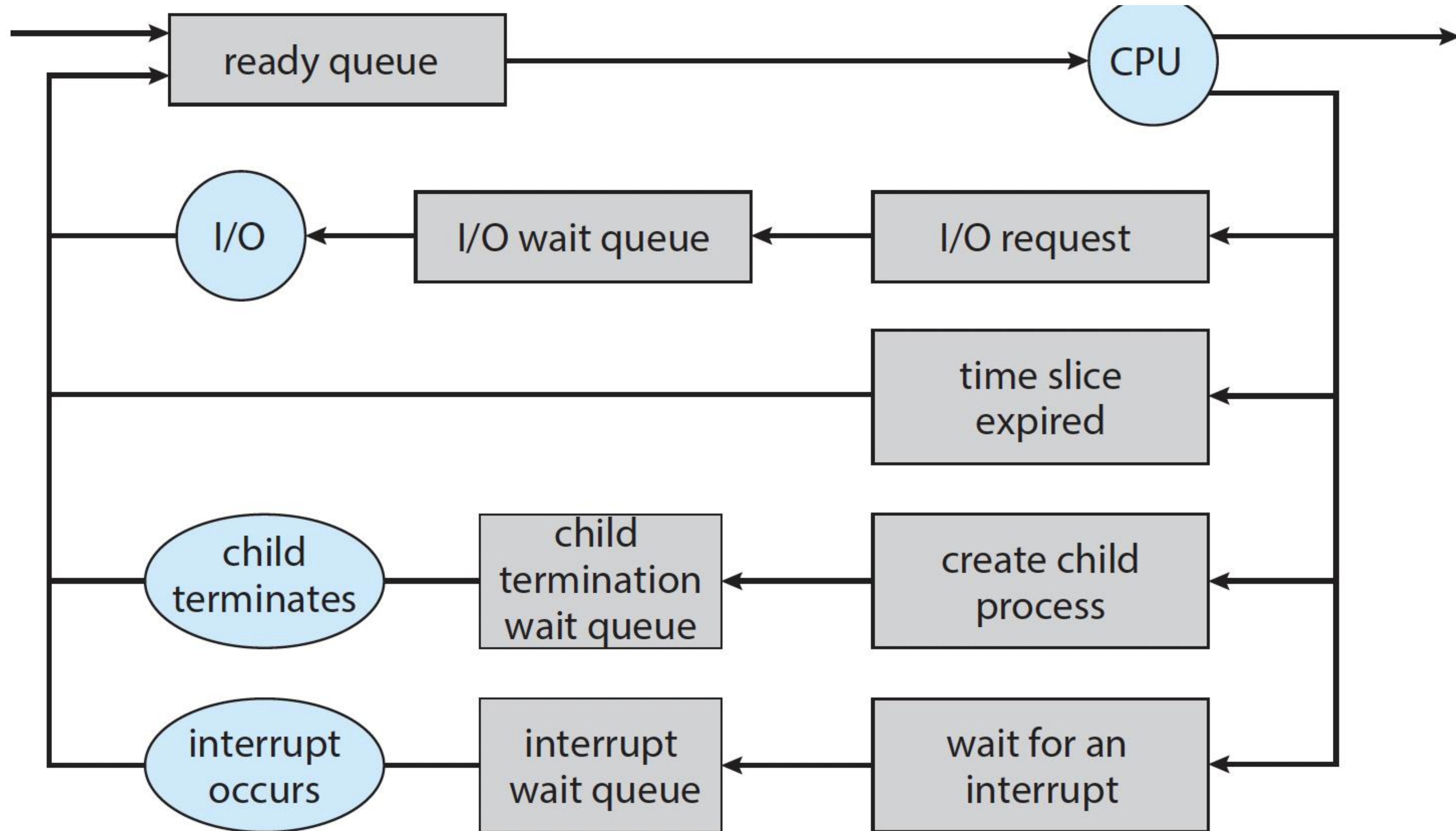
Process Queues

- System maintains several queues for scheduling processes.
- **Ready queue →** set of all processes residing in main memory, ready and waiting to execute.
- **Wait queues →** set of processes waiting for an event (i.e. I/O).
- Processes migrate among the various queues

Process Queues



Process Queues



Context Switch

Context Switch

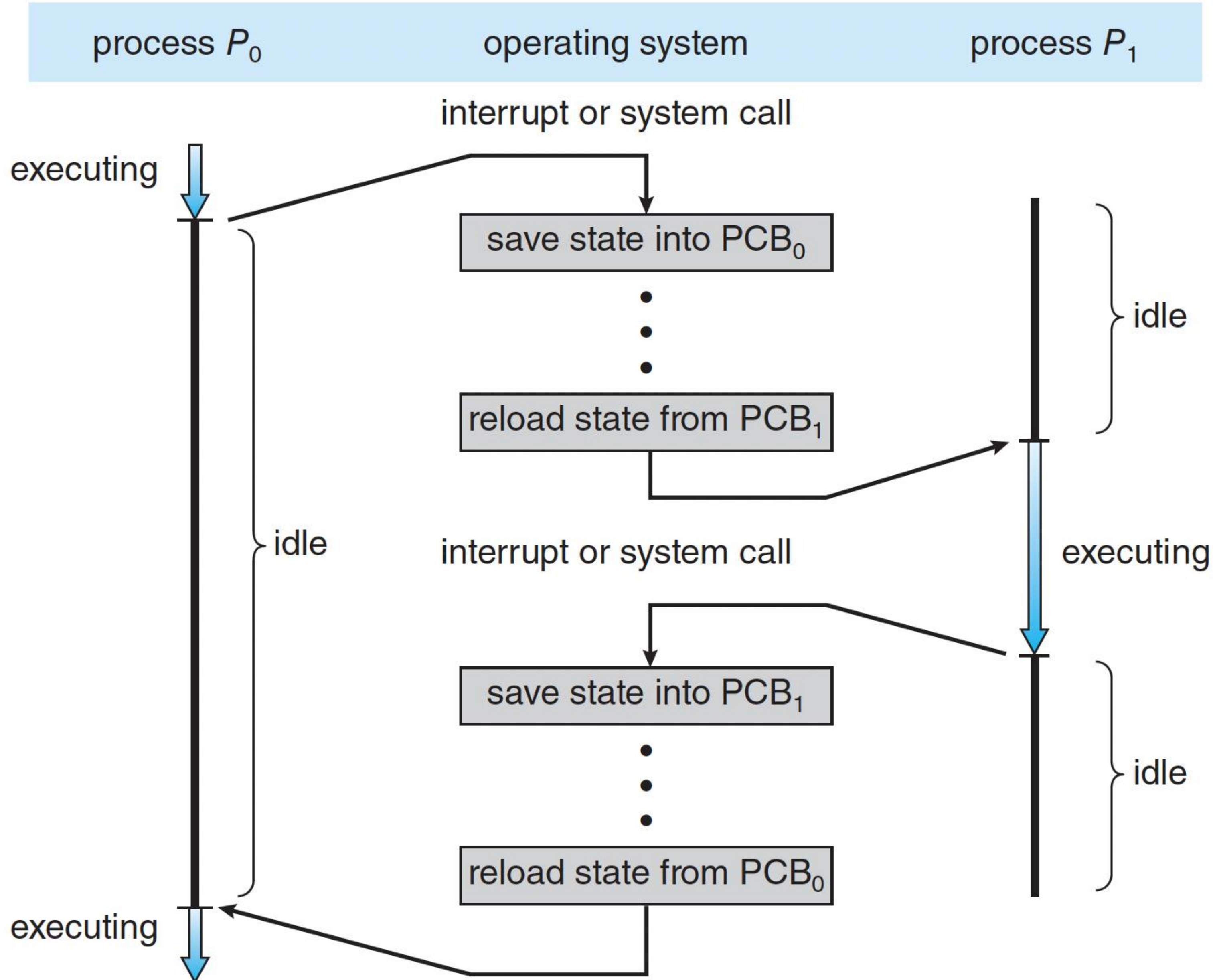
- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a context switch.

- Context of a process represented in the PCB.
    - Includes the value of the CPU registers, the process state, and memory-management information.

- Context-switch time is overhead; the system does no useful work while switching.
    - The more complex the OS and the PCB → longer the context switch.

- Time dependent on hardware support.
    - Some hardware provides multiple sets of registers per CPU.
    - Multiple contexts loaded at once.

Context Switch



Multi-tasking in Mobile

Multi-tasking in Mobile

- Mobile devices are constrained!
  - Early versions of iOS did not provide user-application multitasking.
  - Only one application ran in the foreground while all other were suspended.

- iOS 4 onwards, Apple provided a limited form of multitasking.
  - One foreground application appearing on the display.
  - Multiple background applications remaining in memory (not suspended).

- Later, split-screen allowed running two foreground apps at the same time.

- Android has supported multitasking.

Multi-tasking in Mobile

- An application that requires processing while in the background must use a service, a separate application component that runs on behalf of the background process.

- Example → A streaming audio application.
- If the application moves to the background, the service continues to send audio data to the audio device driver on behalf of the background application.
- The service will continue to run even if the background application is suspended.

Process Creation

Process Creation

- Parent process create children processes, which, in turn create other processes.
  - Like forming a tree of processes.
- A Process identified and managed via a process identifier (pid).

- Resource sharing options
  - Parent and children share all resources.
  - Children share subset of parent's resources.
  - Parent and child share no resources.

- Execution options
  - Parent and children execute concurrently.
  - Parent waits until children terminate.

Process Creation

- Parent process create children processes, which, in turn create other processes.
  - Like forming a tree of processes.
- A Process identified and managed via a process identifier (pid).

- Resource sharing options
  - Parent and children share all resources.
  - Children share subset of parent's resources.
  - Parent and child share no resources.

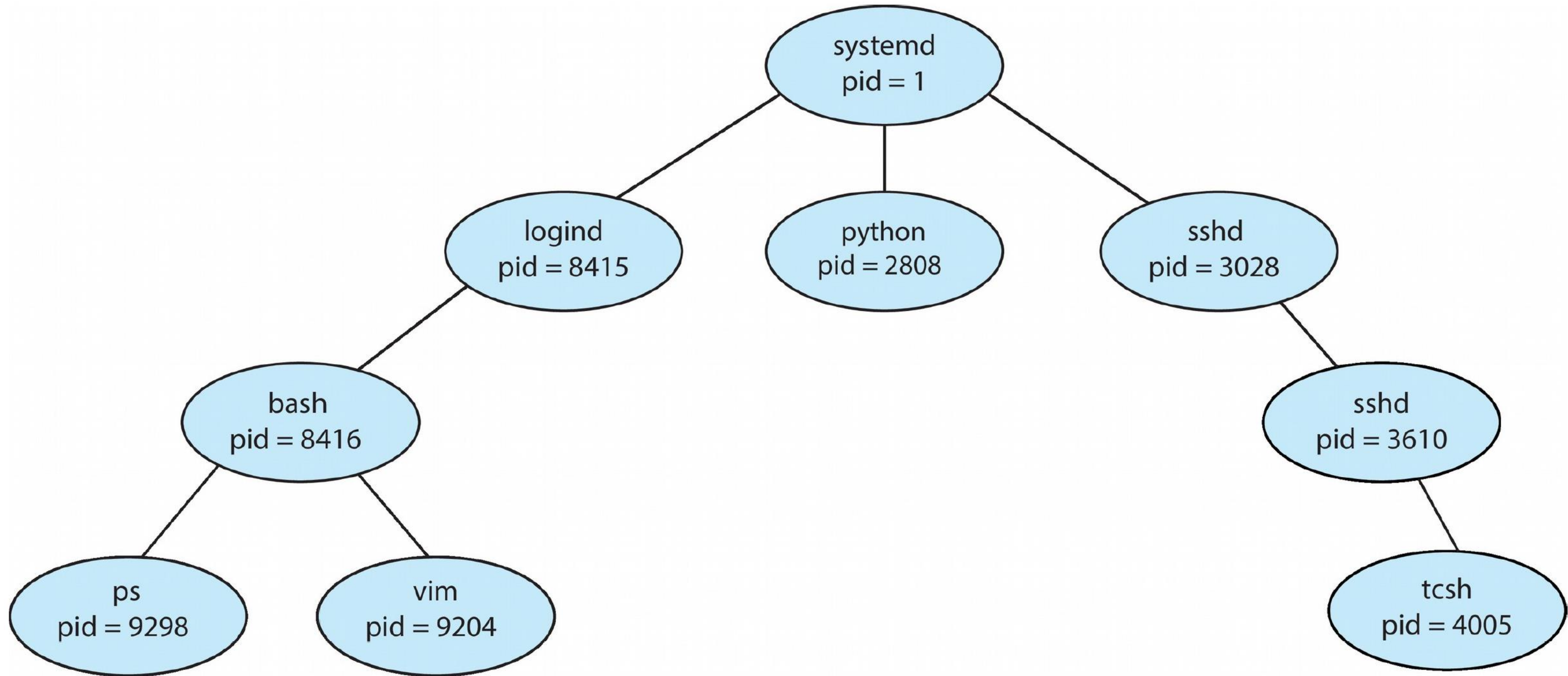
- Execution options
  - Parent and children execute concurrently.
  - Parent waits until children terminate.

Preferred Design?

Process Creation

- There are also two address-space possibilities for the new process:
  - The child process is a duplicate of the parent process (it has the same program and data as the parent).
  - The child process has a new program loaded into it.

Process Tree



Process Creation System Calls

Process Creation System Calls

- How are processes brought into existence?
- **fork(2)**
  - Copy the address space of parent.
  - Return value is PID of child process.
  - Checking value of PID in child process will result in 0 → child process.

- **vfork(2)**
  - Use the parent's address space.
  - Share address space between parent and child.
    - includes data segment (and mapped physical memory)
  - Calling parent blocks until child terminates or calls exec().
  - Faster than fork().

Fork Example

Parent Process

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
      fprintf(stderr, "Fork Failed");
      return 1;
    }
    else if (pid == 0) { /* child process */
      execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
      /* parent will wait for the child to complete */
      wait(NULL);
      printf("Child Complete");
    }

    return 0;
}
```

Fork Example

Parent Process

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
      fprintf(stderr, "Fork Failed");
      return 1;
    }
    else if (pid == 0) { /* child process */
      execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
      /* parent will wait for the child to complete */
      wait(NULL);
      printf("Child Complete");
    }

    return 0;
}
```

Fork Example

Parent Process

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
      fprintf(stderr, "Fork Failed");
      return 1;
    }
    else if (pid == 0) { /* child process */
      execlp("/bin/ls","ls",NULL);
    }
➡   else { /* parent process */
      /* parent will wait for the child to complete */
      wait(NULL);
      printf("Child Complete");
    }

    return 0;
}
```

Child Process

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
      fprintf(stderr, "Fork Failed");
      return 1;
    }
➡   else if (pid == 0) { /* child process */
      execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
      /* parent will wait for the child to complete */
      wait(NULL);
      printf("Child Complete");
    }

    return 0;
}
```

execlp()

Exec Family

- **fork()** allows to create a new child process
  - But child process is forced to run same program.

- **exec()** allows child process to run a different program.

- If you only had **fork()**:
  - You can create children, but they all run the same program as the parent.
  - You can't turn a shell into a ***ls*** process.

- If you only had **exec()**:
  - You could replace your own program with another, but you couldn't keep a parent shell.
  - You would just turn the shell into ***ls***, then into ***grep***, etc.

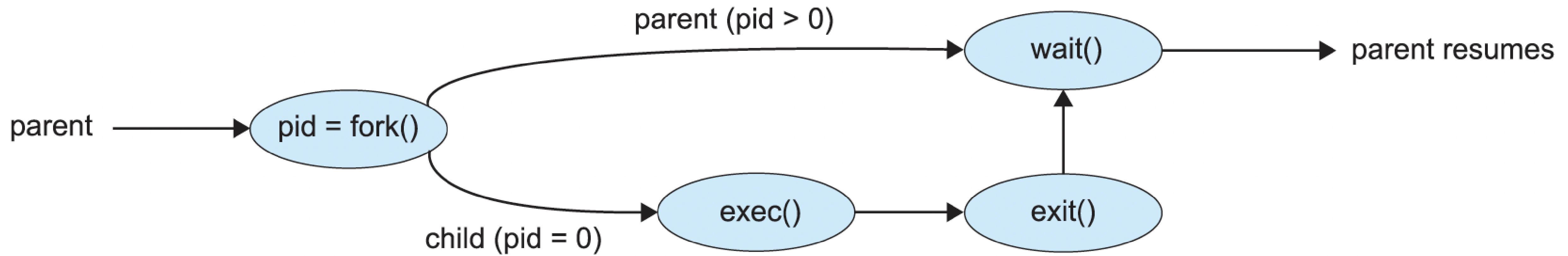
Exec Family Naming Pattern

- All of these ultimately end up in a low-level **execve** system call.

- Letters after exec tell you two things:
- **l vs v**: how you pass arguments.
  - **l →** “list” of arguments: **execl, execlp, execle**.
  - You pass arguments as separate parameters: **execl("/bin/ls","ls","-l",NULL).**
  - **v →**“vector” (array) of arguments: **execv, execvp, execvpe**.
  - You pass **char *argv[]: execvp("ls", argv)**

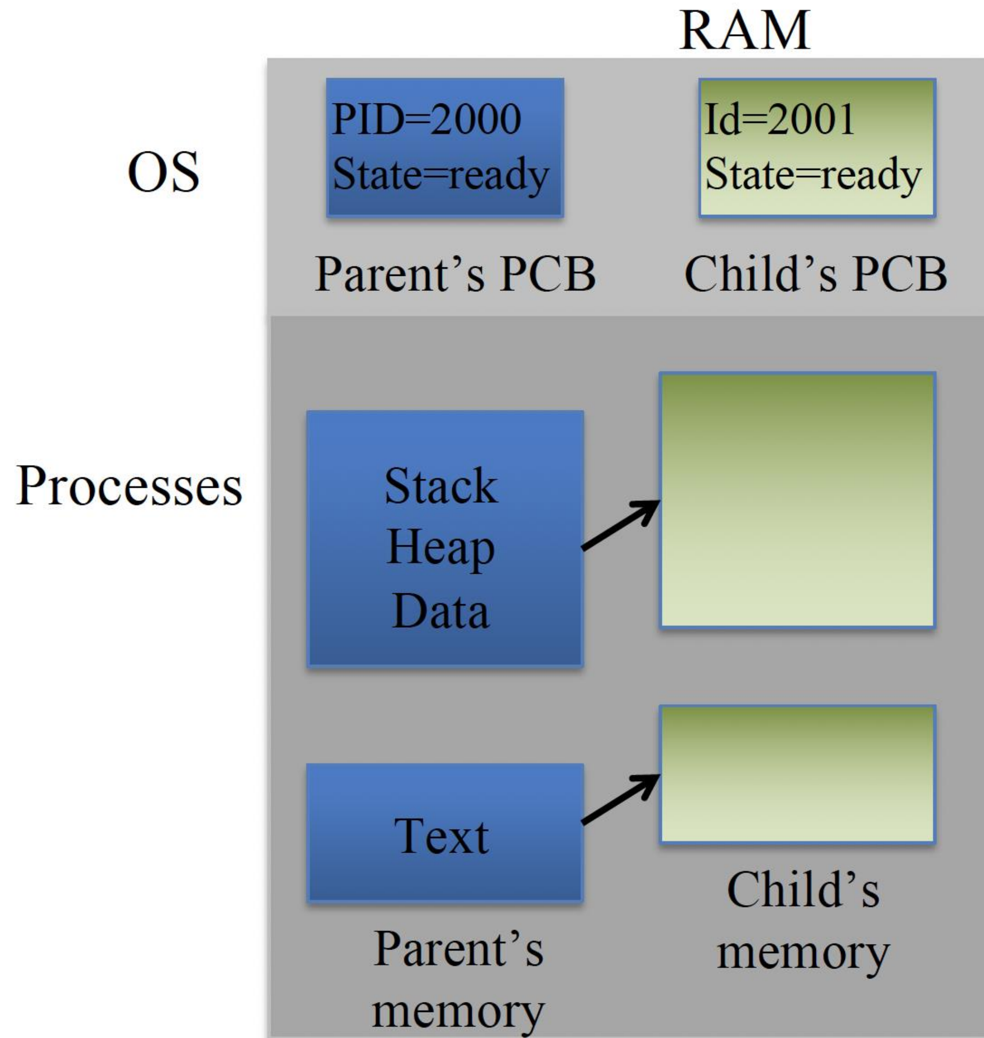
- **p vs no p**: how the program is found.
  - With **p**:→ search using PATH if there’s no / in the name (**execlp, execvp, execvpe**).
  - Without **p**: you must give a path name (absolute or relative): **execl, execv.**

Process Lifecycle



Child Process Memory Layout

Child Process Memory Layout



- A separate PCB for the child process.
- A new process ID (PID) and Parent PID.
- Copy of CPU registers, program counter (PC), stack pointer, etc.
- Copy of memory mappings (code, data, heap, stack segments).
- Copied file descriptors (pointing to same open files).
- Scheduling info, signal handlers, etc.

What if Parent’s Memory is Large?

What if Parent's Memory is Large?

- If the parent's memory is large, the copying cost would be huge.
- In practice, the kernel does not copy all the parent's pages at fork time.
  - Creates a new page table for the child.
- Makes both parent and child map the same physical pages.
  - Marks those shared pages read-only (**Copy-On-Write** (COE) pages).
- If neither parent nor child writes to a page → they keep sharing that physical page, no copy done.
- If either tries to write to a COW page, hardware raises a page fault.
- Kernel allocates a new physical page, copies the old contents,.
  - Now they diverge just for that page.

Discussion

Say you had to redesign the system to avoid Copy-On-Write and you also don't want to copy all the parent process memory to the child process?

Wait

Wait

- Lets a parent wait for their child ?

- **wait(2)**
  - Waits for any child to finish.
  - Blocks the parent until one child exits.
  - Returns the PID of the terminated child.
  - Also return child exit status from argument.

- **waitpid(2)**
  - Waits for the specified child to finish (need to provide child pid).
  - Also return child exit status from argument.
  - Can be configured to either block or non-block.

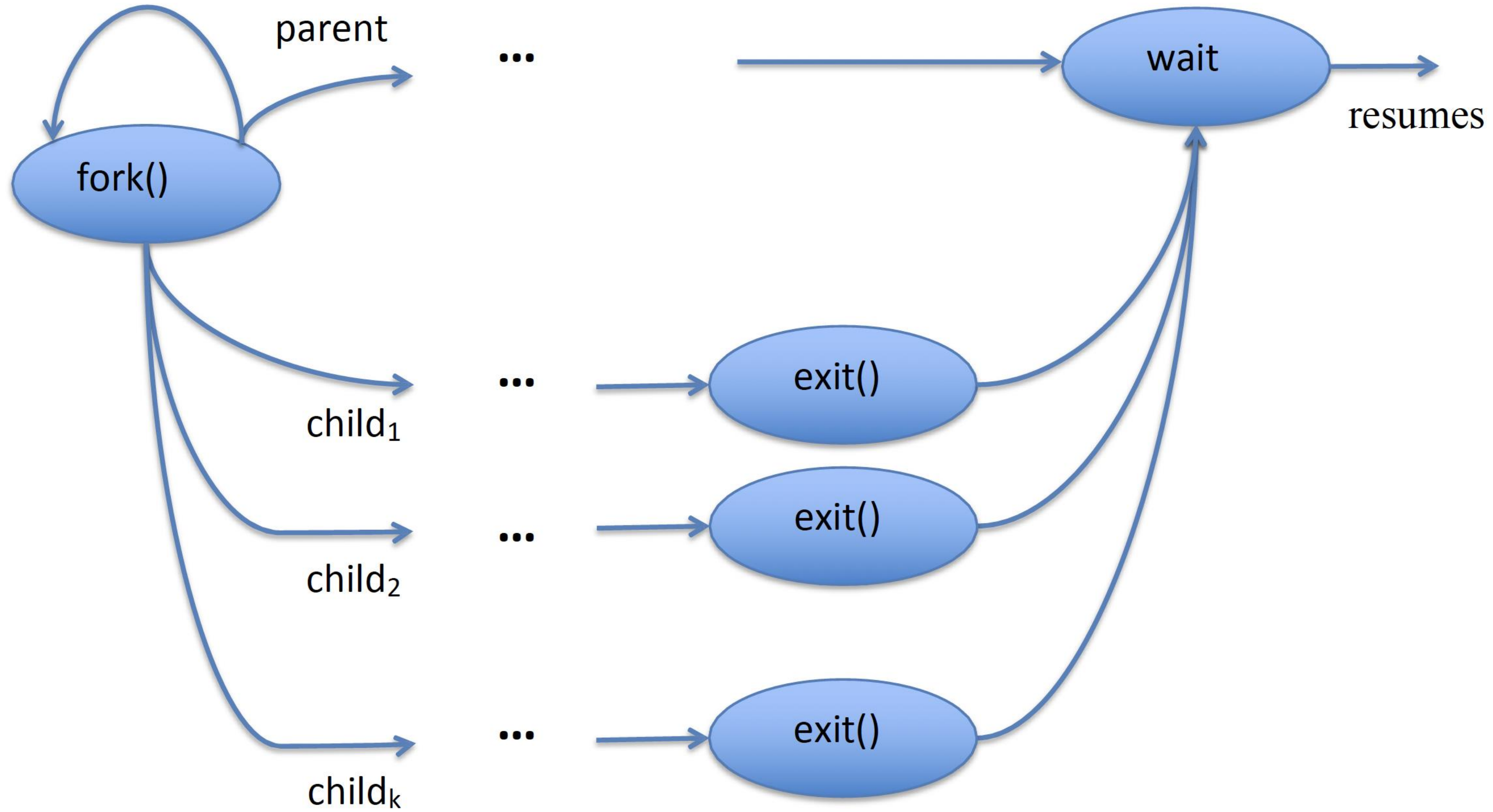
Wait

- Why does the parent process need to wait?

- **Sharing resource**: Parent and child may need to coordinate on using a resource (say disk).
  - Must execute sequentially.

- **Job managing:** Parent may act as a job manager and children are workers.
  - Parent need to know when a job is finished and the status.

Wait



Process Termination

Process Termination

- Process executes last statement and then asks the operating system to delete it using the **exit()** system call.
- Process' resources are deallocated by operating system.
- Parent may terminate the execution of children processes using signals.
  - Child has exceeded allocated resources.
  - Task assigned to child is no longer required.
  - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates.

Cascading Termination

Cascading Termination

- Some operating systems do not allow child to exists if its parent has terminated.
- If a process terminates, then all its children must also be terminated.
- Cascading termination →All children, grandchildren, etc. are terminated.
- The termination is initiated by the operating system.

What happens when a child terminates?

What happens when a child terminates?

- When a child process terminates, it enters the ***zombie*** state:
  - The child's exit code and status are stored in the kernel.
  - The process resources (like its PID entry) are not fully released yet.

- The kernel keeps them until the parent calls wait(2) (or waitpid(2)).

- wait(2) lets the parent:
  - Synchronize with the child's termination, and
  - Reap (collect) the child's exit status —removing the zombie entry.

What happens when a child terminates?

What if a parent never waits?

- The child becomes a zombie after it finishes.
  - When the parent terminates, the child is adopted by **systemd**
  - (PID 1) → called **reparenting**.

- The **systemd** process automatically calls **wait()** to reap it.

- This prevents zombies from accumulating forever.

- A child can continue to run after parent exits early **systemd** won’t force terminating it.

Discussion

How can you design a system where OS by default forces a parent process to wait for all its child process?

Android Process Hierarchy

Android Process Hierarchy

- Because of resource constraints, rather than terminating an arbitrary process, Android follows an importance hierarchy of processes (*most to least below*).
- **Foreground process** → The current process visible on the screen, representing the application the user is currently interacting with.
- **Visible process** → Not directly visible on the foreground but is performing an activity that the foreground process is referring to.
- **Service process** → Similar to a background process but is performing an activity that is apparent to the user (such as streaming music).
- **Background process** → Performing an activity but not apparent to the user.
- **Empty process** → Holds no active components associated with any application.