Operating System Concepts

Lecture 5: Process Management

Omid Ardakanian
oardakan@ualberta.ca
University of Alberta

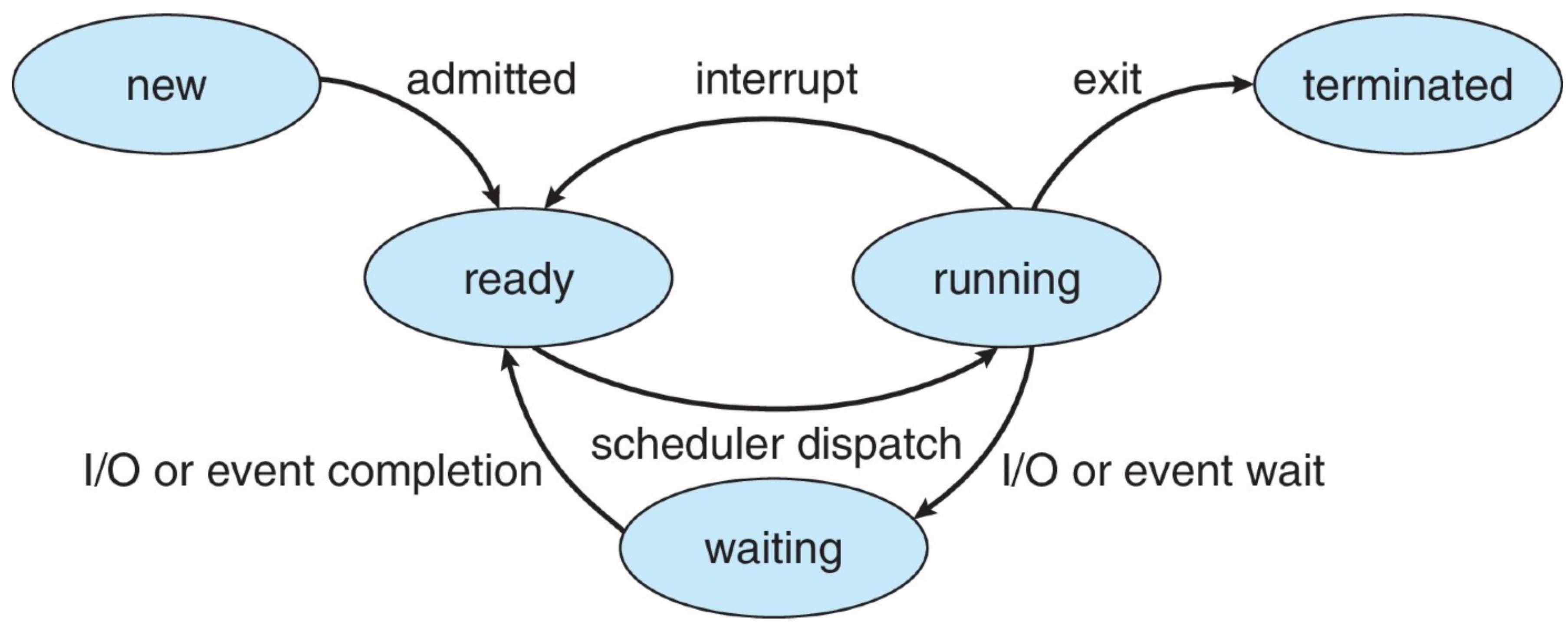
MWF 12:00-12:50 VVC 2 215

Today's class

- Process Abstraction
  - What are the roles of the scheduler and the dispatcher?
  - What happens during context switching?
- Process Control
  - How to create a new process?

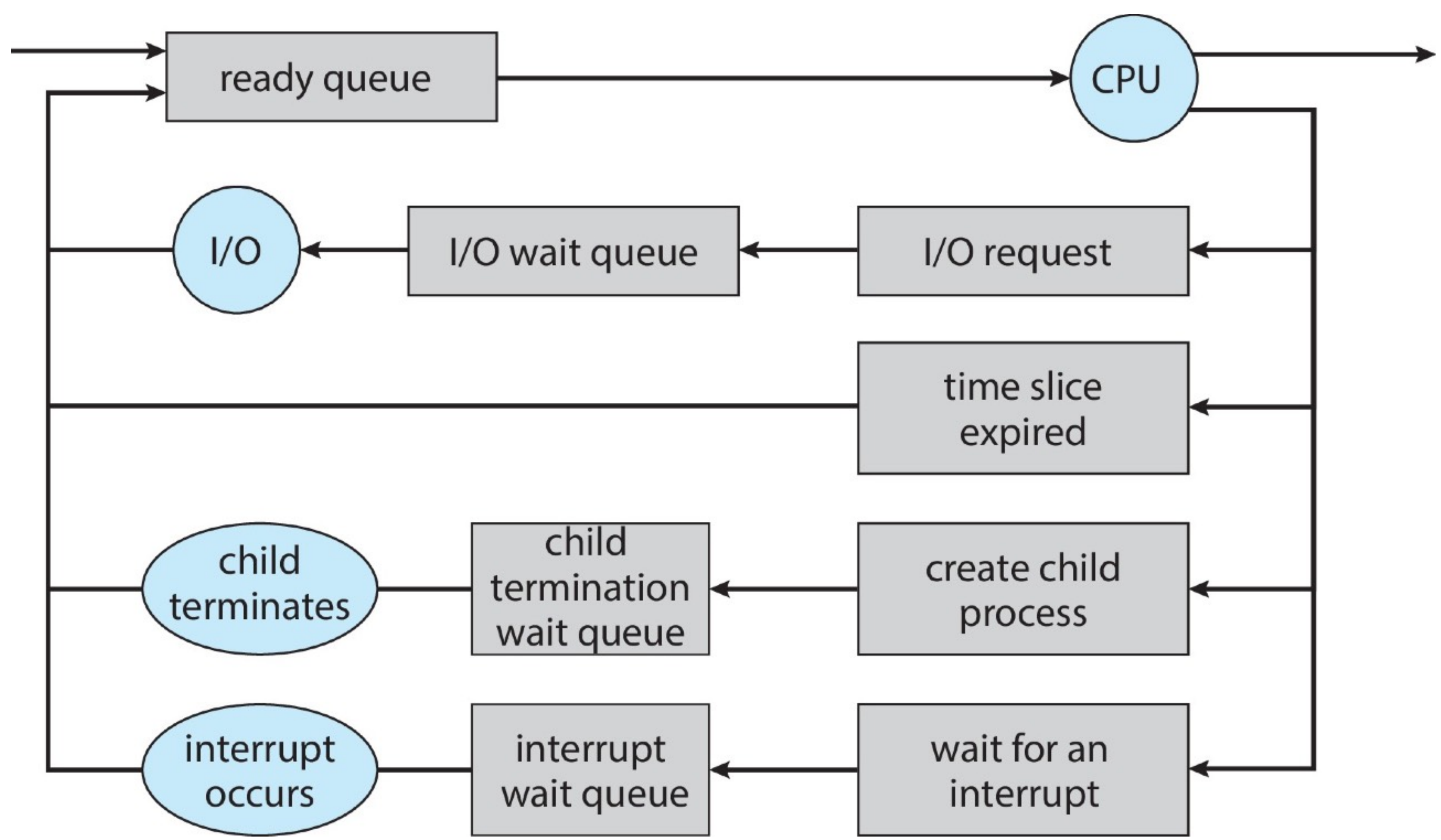
Process lifecycle

- New: being created
- Ready: waiting to be assigned a CPU core
- Running: instructions are running on the CPU core
- Waiting: waiting for some event (e.g., I/O)
- Terminated: finished or aborted



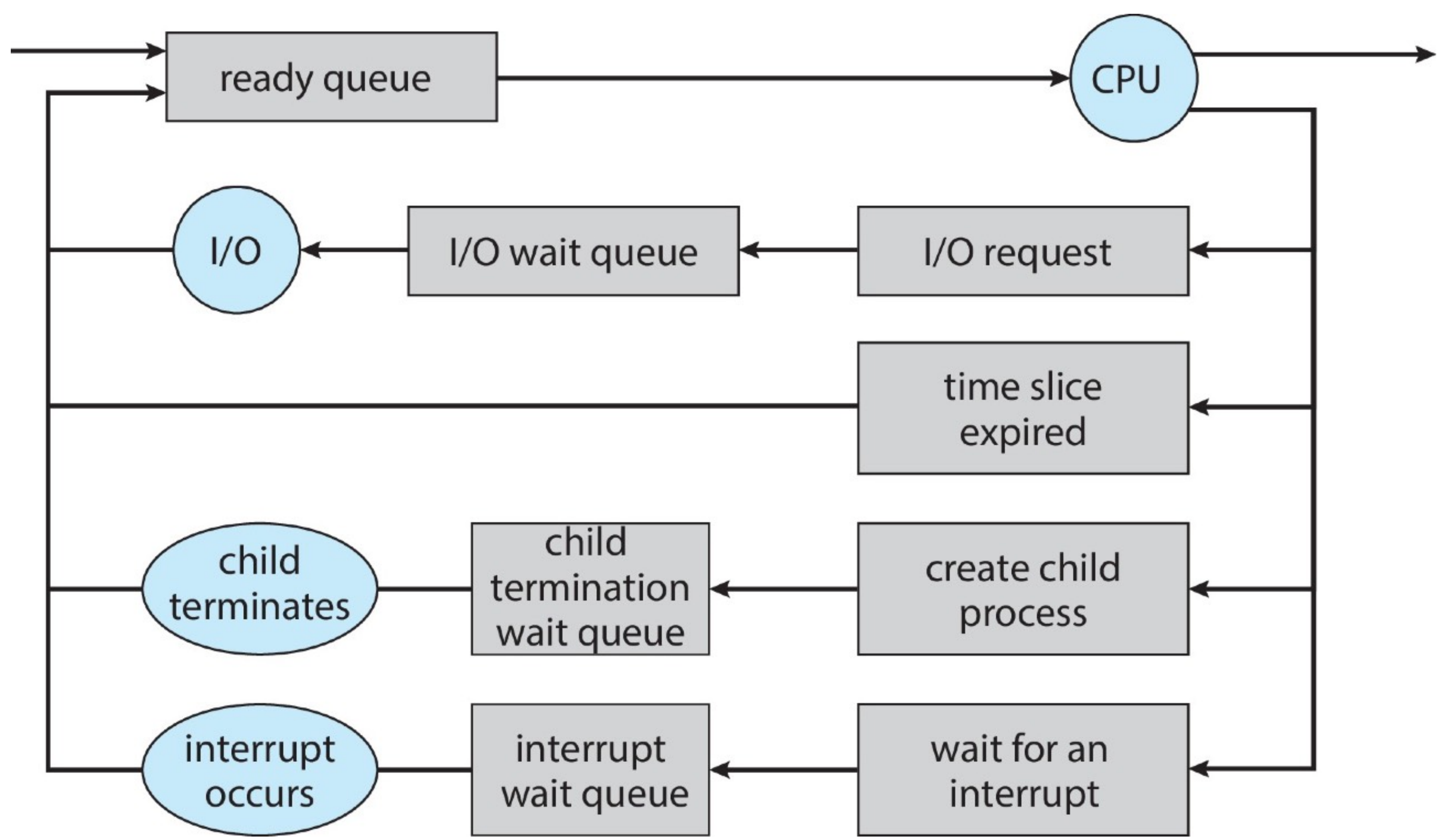
Multiprogramming

- only <u>one</u> process is active at a time (per CPU core)



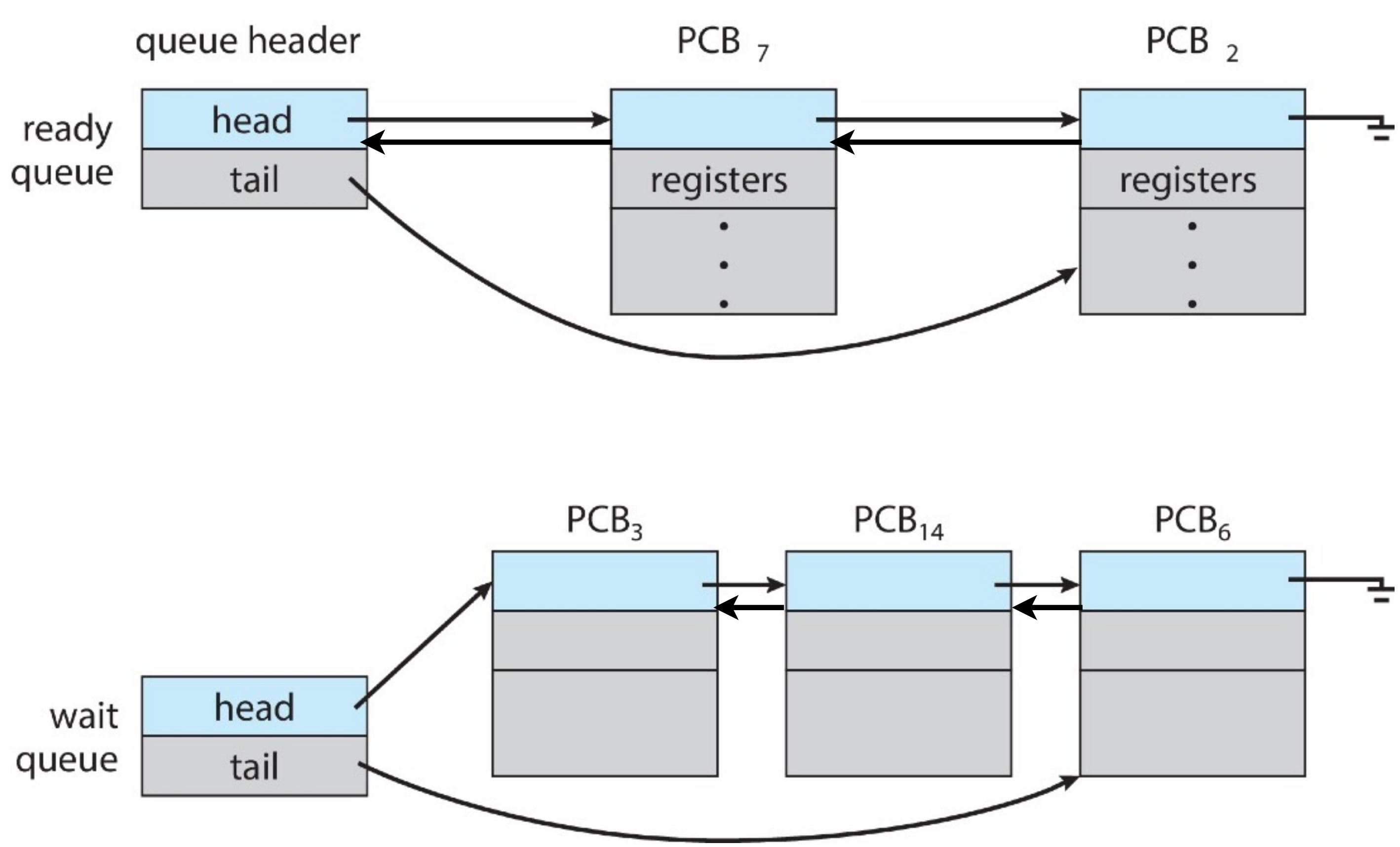
Multiprogramming

- only <u>one</u> process is active at a time (per CPU core)
- OS gives out CPU to different processes in the ready queue
  - it also gives out other resources to them



Scheduling

- the scheduler maintains a data structure (e.g., a **doubly linked list**) of PCBs
  - PCBs are moved from one queue to another queue
  - job queue -> every process in the system
  - ready queue -> processes residing in the main memory, waiting to run on CPU
  - device queue -> processes waiting for a particular I/O device, each device has its own queue



Scheduling

- the scheduler maintains a data structure (e.g., a **doubly linked list**) of PCBs
  - PCBs are moved from one queue to another queue
  - job queue -> every process in the system
  - ready queue -> processes residing in the main memory, waiting to run on CPU
  - device queue -> processes waiting for a particular I/O device, each device has its own queue
- the scheduler selects a process from the ready queue to run
  - the selected process runs on CPU
  - if no process left in the ready queue, the CPU runs an idle process
  - scheduling can be performed for fairness, minimum latency, providing real-time guarantees, etc.

Scheduling

- the scheduler maintains a data structure (e.g., a **doubly linked list**) of PCBs
  - PCBs are moved from one queue to another queue
  - job queue -> every process in the system
  - ready queue -> processes residing in the main memory, waiting to run on CPU
  - device queue -> processes waiting for a particular I/O device, each device has its own queue
- the scheduler selects a process from the ready queue to run
  - the selected process runs on CPU
  - if no process left in the ready queue, the CPU runs an idle process
  - scheduling can be performed for fairness, minimum latency, providing real-time guarantees, etc.
- the (short-term) scheduler makes a decision very fast (<10 ms) and is called very often (e.g., every 100ms)

Context switching

- a context switch is stopping a process and starting another one
  - it is a relatively expensive operation
  - time-sharing systems may do hundreds of context switches per second
  - the **context** includes the value of CPU registers, the process state, and memory management information

Context switching

- a context switch is stopping a process and starting another one
  - it is a relatively expensive operation
  - time-sharing systems may do hundreds of context switches per second
  - the **context** includes the value of CPU registers, the process state, and memory management information
- OS starts executing a process in the ready state by loading hardware registers (PC, SP, etc) from its PCB
  - while a process is running, the CPU modifies the Program Counter (PC), Stack Pointer (SP), registers, etc.

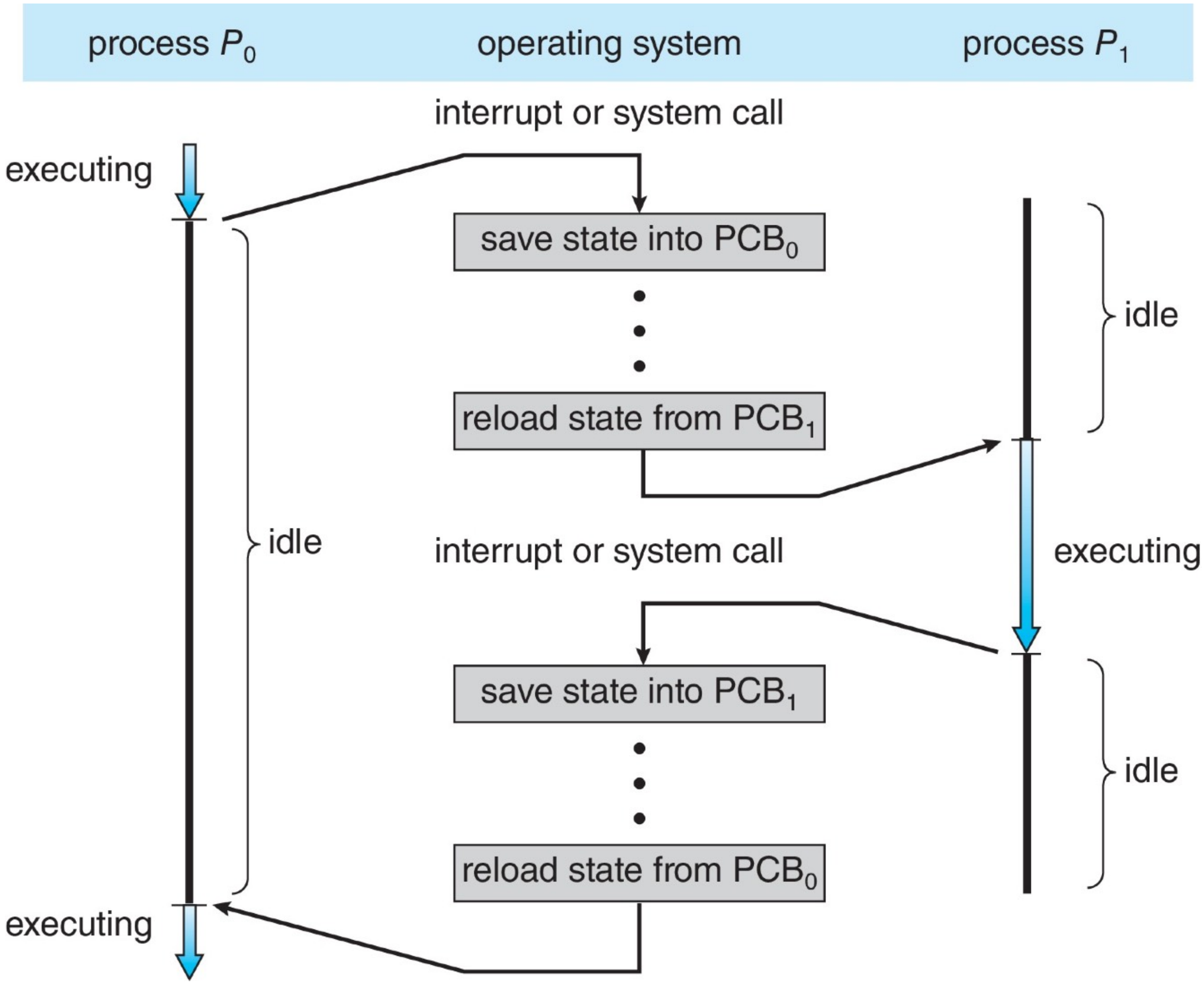
Context switching

- a context switch is stopping a process and starting another one
  - it is a relatively expensive operation
  - time-sharing systems may do hundreds of context switches per second
  - the **context** includes the value of CPU registers, the process state, and memory management information
- OS starts executing a process in the ready state by loading hardware registers (PC, SP, etc) from its PCB
  - while a process is running, the CPU modifies the Program Counter (PC), Stack Pointer (SP), registers, etc.
- when OS stops executing a process, it saves the values of the registers (PC, SP, etc.) into its PCB
  - so that they can be restored the next time the process is selected for running on the CPU

Context switching

- a context switch is stopping a process and starting another one
  - it is a relatively expensive operation
  - time-sharing systems may do hundreds of context switches per second
  - the **context** includes the value of CPU registers, the process state, and memory management information
- OS starts executing a process in the ready state by loading hardware registers (PC, SP, etc) from its PCB
  - while a process is running, the CPU modifies the Program Counter (PC), Stack Pointer (SP), registers, etc.
- when OS stops executing a process, it saves the values of the registers (PC, SP, etc.) into its PCB
  - so that they can be restored the next time the process is selected for running on the CPU
- context switching is pure **overhead,** i.e., the system does no useful work
  - the cost of a context switch and the time between switches are closely related
    - but fast context switching is necessary for responsiveness
  - OS must balance the context switch frequency with the scheduling requirement

Context switching



Scheduler and dispatcher

Scheduler and dispatcher

- the scheduler selects a process to run next

Scheduler and dispatcher

- the scheduler selects a process to run next
- the dispatcher makes it happen
  - performs context switching
  - switches to the user mode
  - jumps to the proper location in the user program

Process Management

Process management system calls in UNIX

Process management system calls in UNIX

- `getpid( )` returns the current PID

Process management system calls in UNIX

- `getpid( )` returns the current PID
- `fork( )` copies the current process (a new PID is assigned to the child process)

Process management system calls in UNIX

- `getpid( )` returns the current PID
- `fork( )` copies the current process (a new PID is assigned to the child process)
- `exec( )` loads a new binary file into memory (without changing its PID)

Process management system calls in UNIX

- `getpid( )` returns the current PID
- `fork( )` copies the current process (a new PID is assigned to the child process)
- `exec( )` loads a new binary file into memory (without changing its PID)
- `wait( )` waits until one of its child processes terminates

Process management system calls in UNIX

- `getpid( )` returns the current PID
- `fork( )` copies the current process (a new PID is assigned to the child process)
- `exec( )` loads a new binary file into memory (without changing its PID)
- `wait( )` waits until <u>one of its child processes</u> terminates
- `waitpid( )` waits until <u>the specified child process</u> terminates

Process management system calls in UNIX

- `getpid( )` returns the current PID
- `fork( )` copies the current process (a new PID is assigned to the child process)
- `exec( )` loads a new binary file into memory (without changing its PID)
- `wait( )` waits until <u>one of its child processes</u> terminates
- `waitpid( )` waits until <u>the specified child process</u> terminates
- `exit( )` terminates a process

Process management system calls in UNIX

- `getpid( )` returns the current PID
- `fork( )` copies the current process (a new PID is assigned to the child process)
- `exec( )` loads a new binary file into memory (without changing its PID)
- `wait( )` waits until <u>one of its child processes</u> terminates
- `waitpid( )` waits until <u>the specified child process</u> terminates
- `exit( )` terminates a process
- `kill( )` sends a signal (interrupt-like notification) to another process

Process management system calls in UNIX

- `getpid( )` returns the current PID
- `fork( )` copies the current process (a new PID is assigned to the child process)
- `exec( )` loads a new binary file into memory (without changing its PID)
- `wait( )` waits until <u>one of its child processes</u> terminates
- `waitpid( )` waits until <u>the specified child process</u> terminates
- `exit( )` terminates a process
- `kill( )` sends a signal (interrupt-like notification) to another process
- `pause( )` causes the calling process to sleep until a signal is delivered that either terminates the process or causes the invocation of a signal-catching function

Process management system calls in UNIX

- `getpid( )` returns the current PID
- `fork( )` copies the current process (a new PID is assigned to the child process)
- `exec( )` loads a new binary file into memory (without changing its PID)
- `wait( )` waits until <u>one of its child processes</u> terminates
- `waitpid( )` waits until <u>the specified child process</u> terminates
- `exit( )` terminates a process
- `kill( )` sends a signal (interrupt-like notification) to another process
- `pause( )` causes the calling process to sleep until a signal is delivered that either terminates the process or causes the invocation of a signal-catching function
- `nanosleep( )` suspends execution of a process for at least the specified time (can be interrupted by a signal that triggers the invocation of a handler)

Process management system calls in UNIX

- `getpid( )` returns the current PID
- `fork( )` copies the current process (a new PID is assigned to the child process)
- `exec( )` loads a new binary file into memory (without changing its PID)
- `wait( )` waits until <u>one of its child processes</u> terminates
- `waitpid( )` waits until <u>the specified child process</u> terminates
- `exit( )` terminates a process
- `kill( )` sends a signal (interrupt-like notification) to another process
- `pause( )` causes the calling process to sleep until a signal is delivered that either terminates the process or causes the invocation of a signal-catching function
- `nanosleep( )` suspends execution of a process for at least the specified time (can be interrupted by a signal that triggers the invocation of a handler)
- `sigaction( )` sets handlers for signals

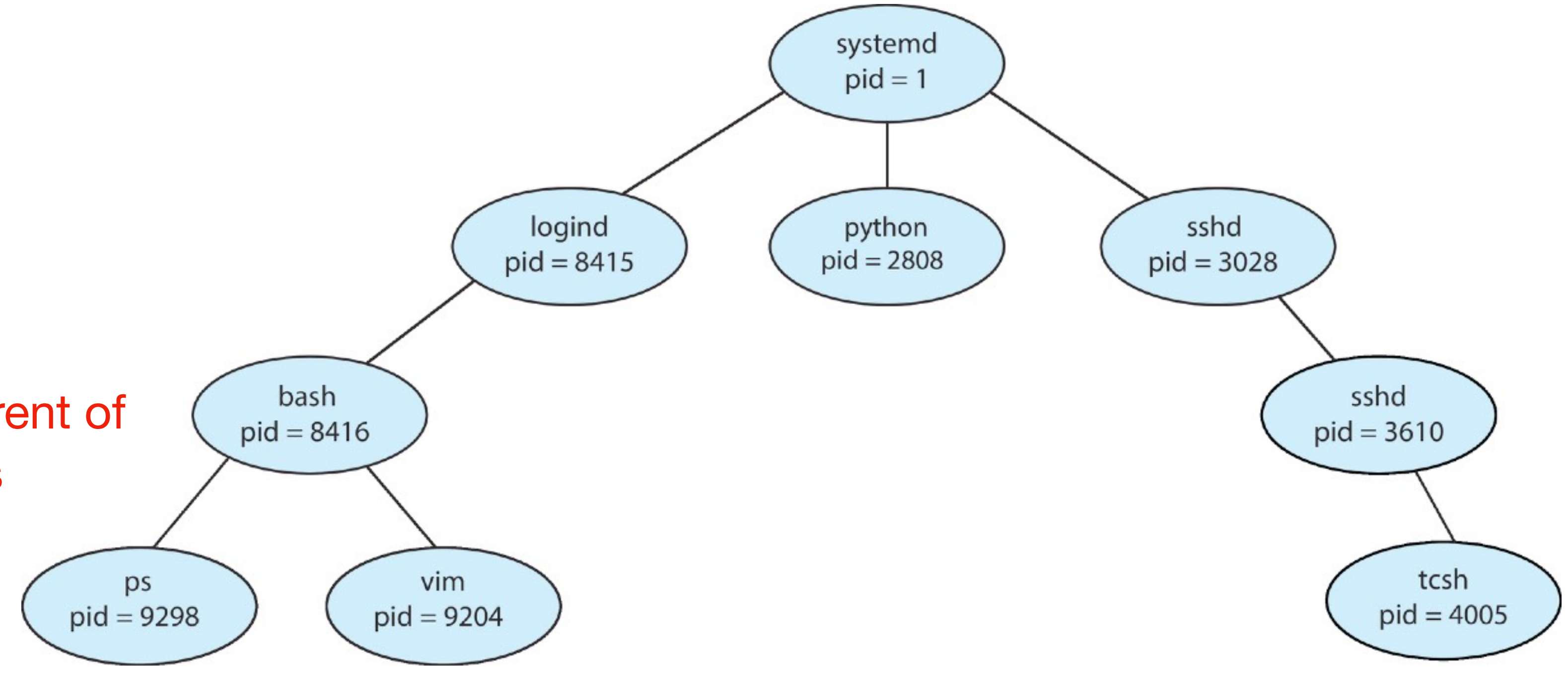
Creating a process with the fork system call

- the `fork( )` system call creates a **child** process which inherits a copy of its **parent**'s memory, file descriptors, CPU registers, etc.

Creating a process with the fork system call

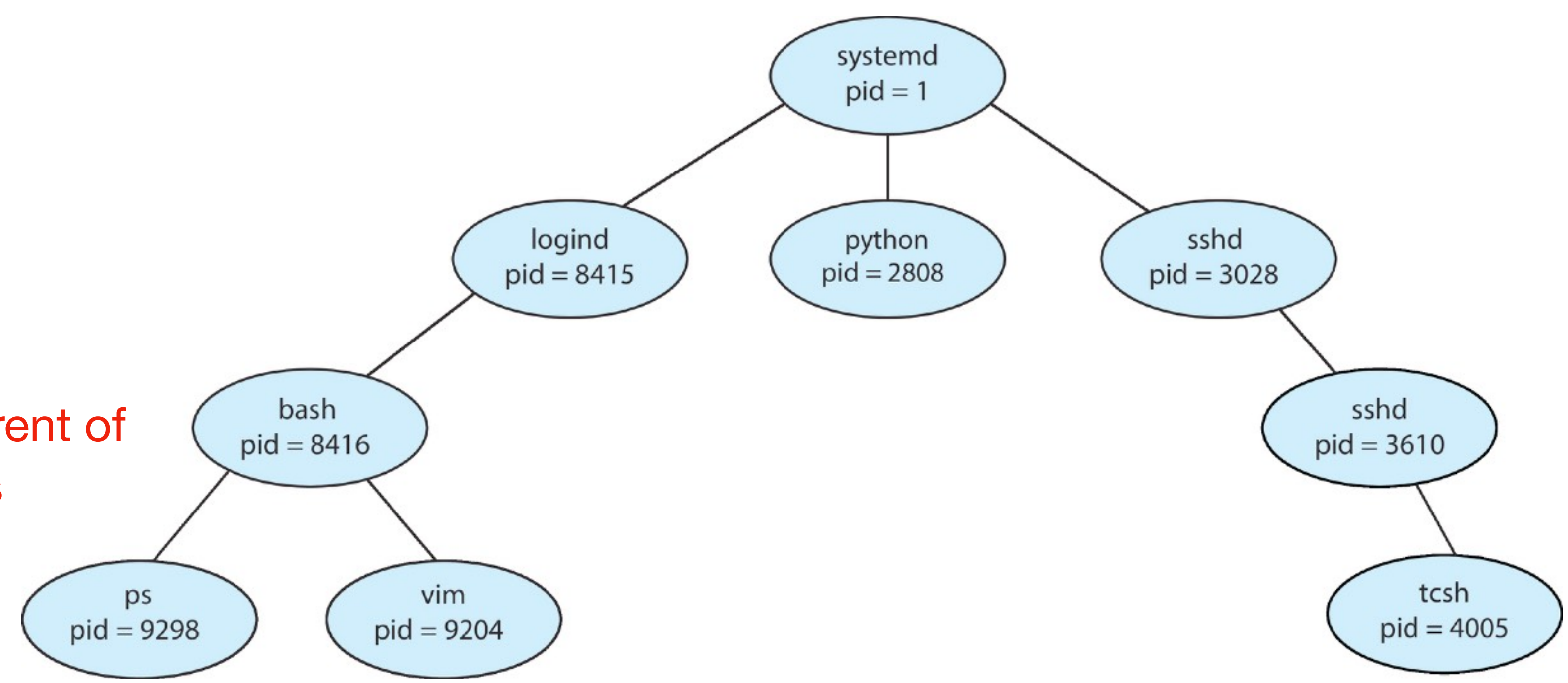
- the `fork( )` system call creates a **child** process which inherits a copy of its **parent**'s memory, file descriptors, CPU registers, etc.

the `systemd` process is the root of this tree (parent of all user processes created when the system has booted); it has PID =1



Creating a process with the fork system call

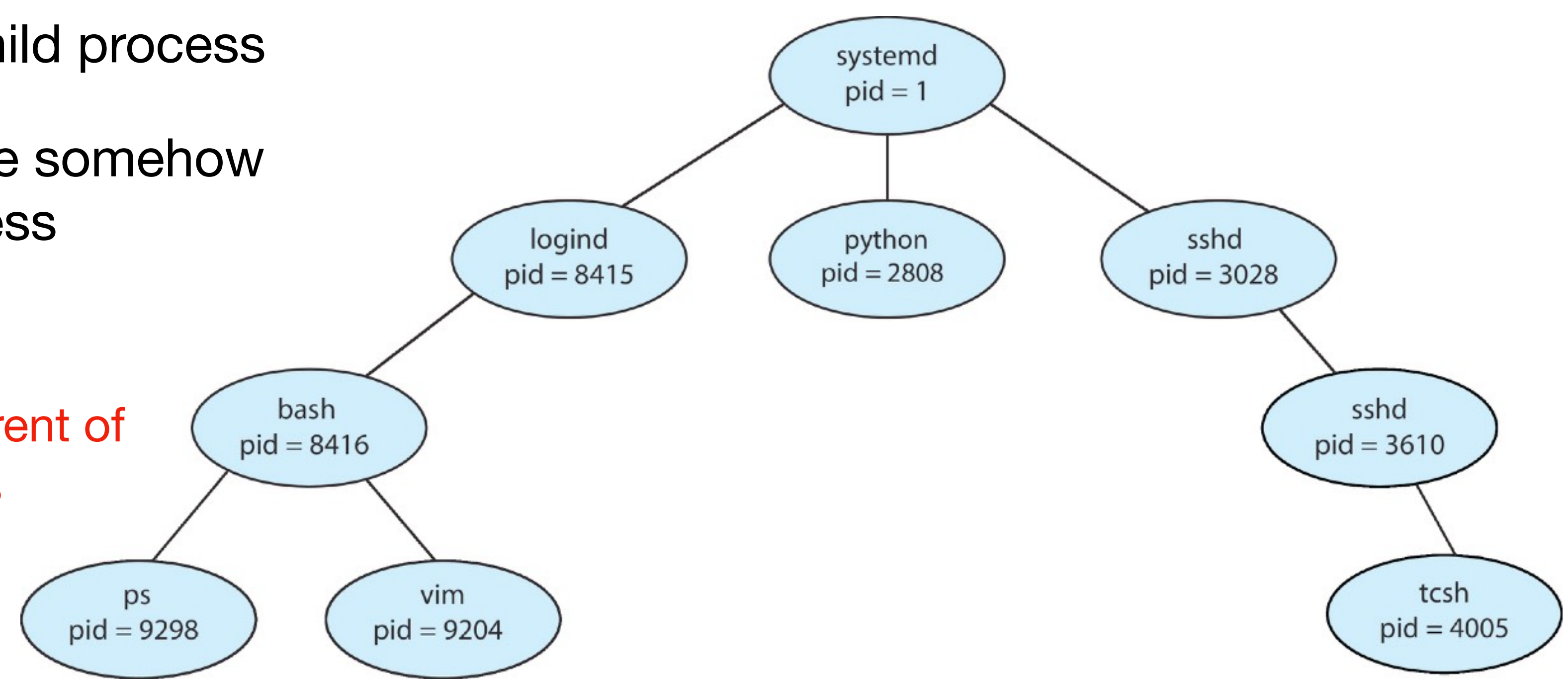
- the `fork( )` system call creates a **child** process which inherits a copy of its **parent**'s memory, file descriptors, CPU registers, etc.
- both parent and child processes execute from the instruction following `fork( )`
  - either the child or the parent might run first



the `systemd` process is the root of this tree (parent of all user processes created when the system has booted); it has PID =1

Creating a process with the fork system call

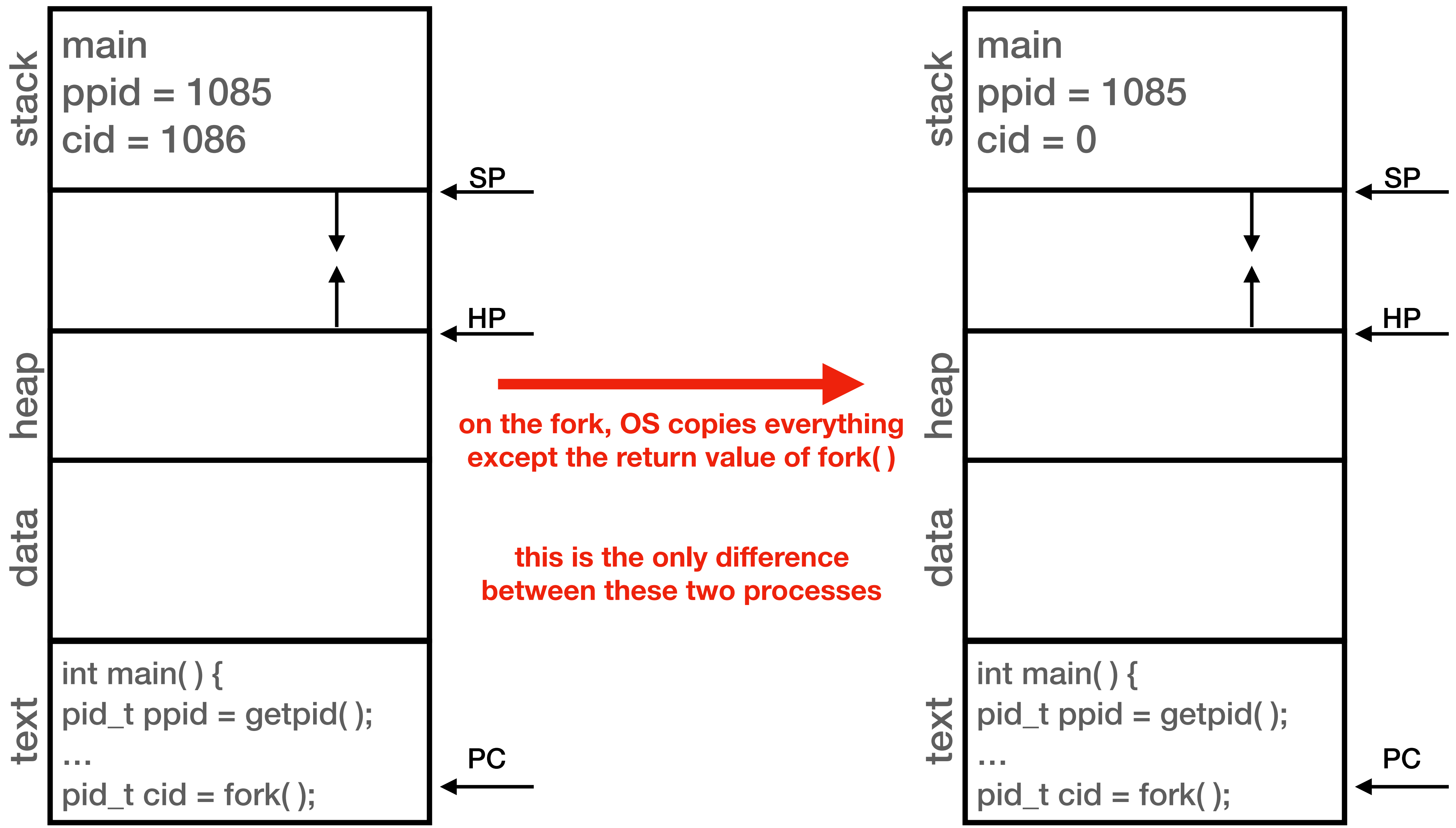
- the `fork( )` system call creates a **child** process which inherits a copy of its **parent**'s memory, file descriptors, CPU registers, etc.
- both parent and child processes execute from the instruction following `fork( )`
  - either the child or the parent might run first
- the return value from `fork( )` is of type pid_t (like an integer)
  - when > 0: running in (original) parent process and the return value is pid of new child
  - when = 0: running in new child process
  - when < 0: error! must handle somehow still running in original process



the `systemd` process is the root of this tree (parent of all user processes created when the system has booted); it has PID =1

What happens on a fork?

```
pid_t cid = fork( );
```



Common problems with fork

Common problems with fork

the `fork( )` system call is

Common problems with fork

the `fork( )` system call is

- inefficient and slow
  - the cost of copying the entire address space of a process is high

Common problems with fork

the `fork( )` system call is

- inefficient and slow
  - the cost of copying the entire address space of a process is high
- insecure
  - the parent process must explicitly remove states that the child process does not need (scrubbing secrets from memory)

Common problems with fork

the `fork( )` system call is

- inefficient and slow
  - the cost of copying the entire address space of a process is high
- insecure
  - the parent process must explicitly remove states that the child process does not need (scrubbing secrets from memory)
- not thread-safe
  - the child process created by the fork system call will have a single thread only (a copy of the calling thread)
  - **Problem**: one thread doing memory allocation and holding a **heap lock**, while another thread forks. Any attempt to allocate memory in the child (and thus acquire the same lock) will immediately **deadlock** waiting for an unlock operation that will never happen
  - **Solution**: not using fork in a multithreaded process, or calling exec immediately afterwards

Program loading with the exec system call

- the `exec( )` system call allows a process to load a different program and start execution at main

Program loading with the exec system call

- the `exec( )` system call allows a process to load a different program and start execution at main
- it allows a process to specify the number of arguments (argc) and the string argument array (argv)

Program loading with the exec system call

- the `exec( )` system call allows a process to load a different program and start execution at main
- it allows a process to specify the number of arguments (argc) and the string argument array (argv)
- if the call is successful, the same process runs a different program

Program loading with the exec system call

- the `exec( )` system call allows a process to load a different program and start execution at main
- it allows a process to specify the number of arguments (argc) and the string argument array (argv)
- if the call is successful, the same process runs a different program
- code, data, stack and heap sections are overwritten

Program loading with the exec system call

- the `exec( )` system call allows a process to load a different program and start execution at main
- it allows a process to specify the number of arguments (argc) and the string argument array (argv)
- if the call is successful, the same process runs a different program
- code, data, stack and heap sections are overwritten
- in most cases we call `exec( )` after calling `fork( )`
  - hence, the memory copied during `fork( )` is useless
  - the `vfork( )` system call allows for creating a process without creating an identical memory image; in this case child process must call `exec( )` immediately

Waiting for the child process to terminate

Waiting for the child process to terminate

- the parent can execute concurrently with its children or can wait until some or all of them terminate

Waiting for the child process to terminate

- the parent can execute concurrently with its children or can wait until some or all of them terminate
- the `wait( )` system call enables the parent process to wait for the child process to terminate and receive its return value

Waiting for the child process to terminate

- the parent can execute concurrently with its children or can wait until some or all of them terminate
- the `wait( )` system call enables the parent process to wait for the child process to terminate and receive its return value
  - it puts the parent to sleep waiting for a child's result

Waiting for the child process to terminate

- the parent can execute concurrently with its children or can wait until some or all of them terminate
- the `wait( )` system call enables the parent process to wait for the child process to terminate and receive its return value
  - it puts the parent to sleep waiting for a child's result
  - when a child calls `exit( )`, the OS unblocks the parent and returns the value passed by `exit( )` as a result of the wait call (along with the pid of the child)

Waiting for the child process to terminate

- the parent can execute concurrently with its children or can wait until some or all of them terminate
- the `wait( )` system call enables the parent process to wait for the child process to terminate and receive its return value
  - it puts the parent to sleep waiting for a child's result
  - when a child calls `exit( )`, the OS unblocks the parent and returns the value passed by `exit( )` as a result of the wait call (along with the pid of the child)
  - if there are no children alive, `wait( )` returns immediately

Waiting for the child process to terminate

- the parent can execute concurrently with its children or can wait until some or all of them terminate
- the `wait( )` system call enables the parent process to wait for the child process to terminate and receive its return value
  - it puts the parent to sleep waiting for a child's result
  - when a child calls `exit( )`, the OS unblocks the parent and returns the value passed by `exit( )` as a result of the wait call (along with the pid of the child)
  - if there are no children alive, `wait( )` returns immediately
  - also, if there are zombies waiting for their parents, `wait( )` returns one of the values immediately (and deallocates the zombie)

Waiting for the child process to terminate

- the parent can execute concurrently with its children or can wait until some or all of them terminate
- the `wait( )` system call enables the parent process to wait for the child process to terminate and receive its return value
  - it puts the parent to sleep waiting for a child's result
  - when a child calls `exit( )`, the OS unblocks the parent and returns the value passed by `exit( )` as a result of the wait call (along with the pid of the child)
  - if there are no children alive, `wait( )` returns immediately
  - also, if there are zombies waiting for their parents, `wait( )` returns one of the values immediately (and deallocates the zombie)