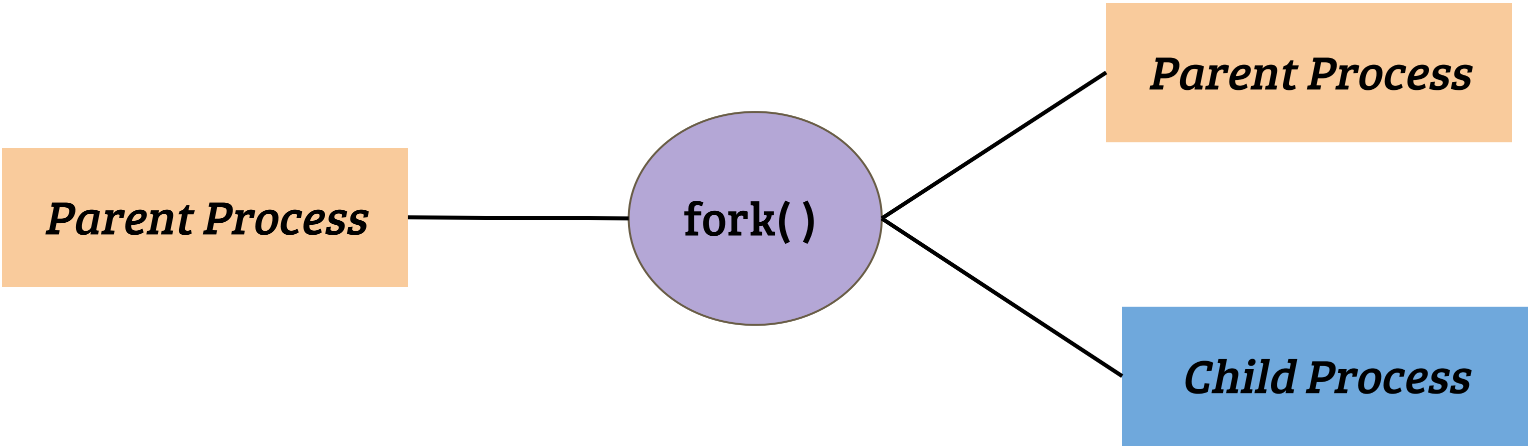
Operating Systems

Process API and system calls

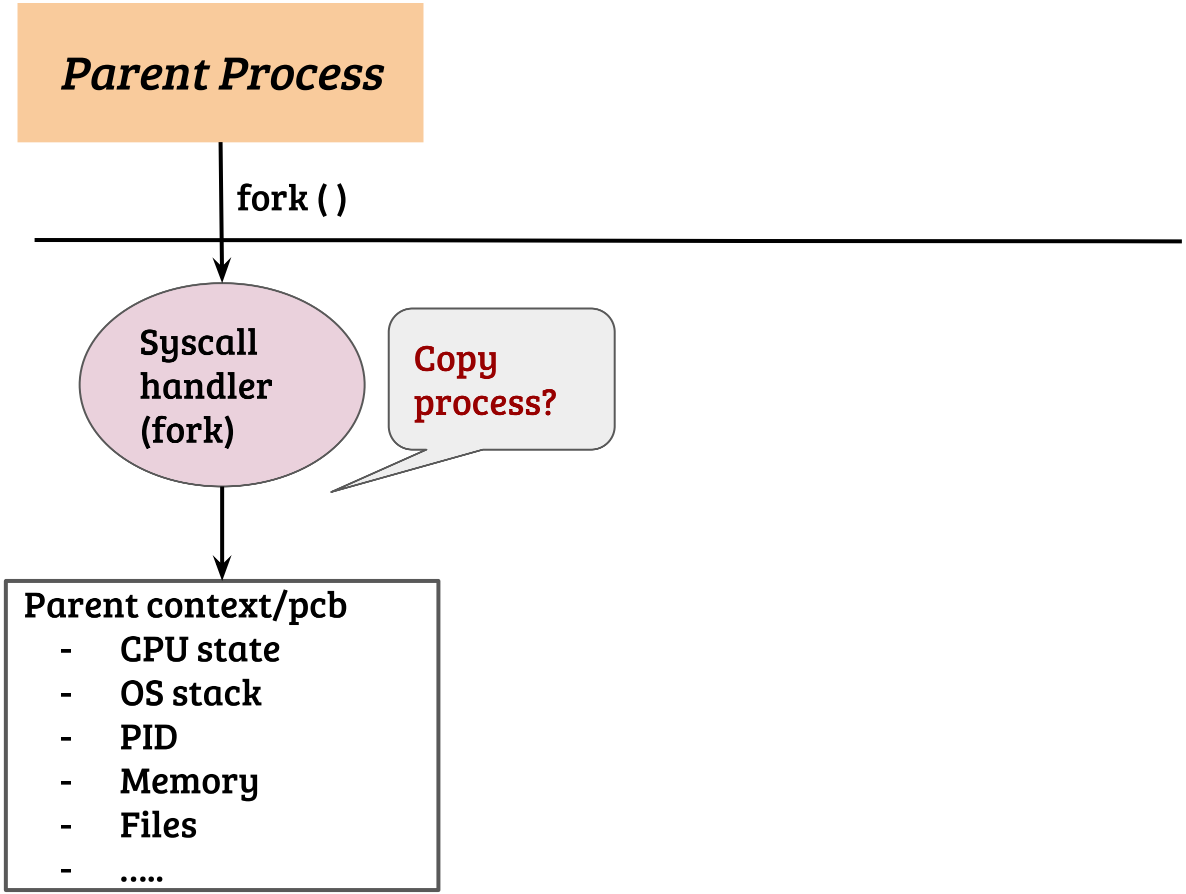
Debadatta Mishra, CSE, IITK

Process creation - fork( )

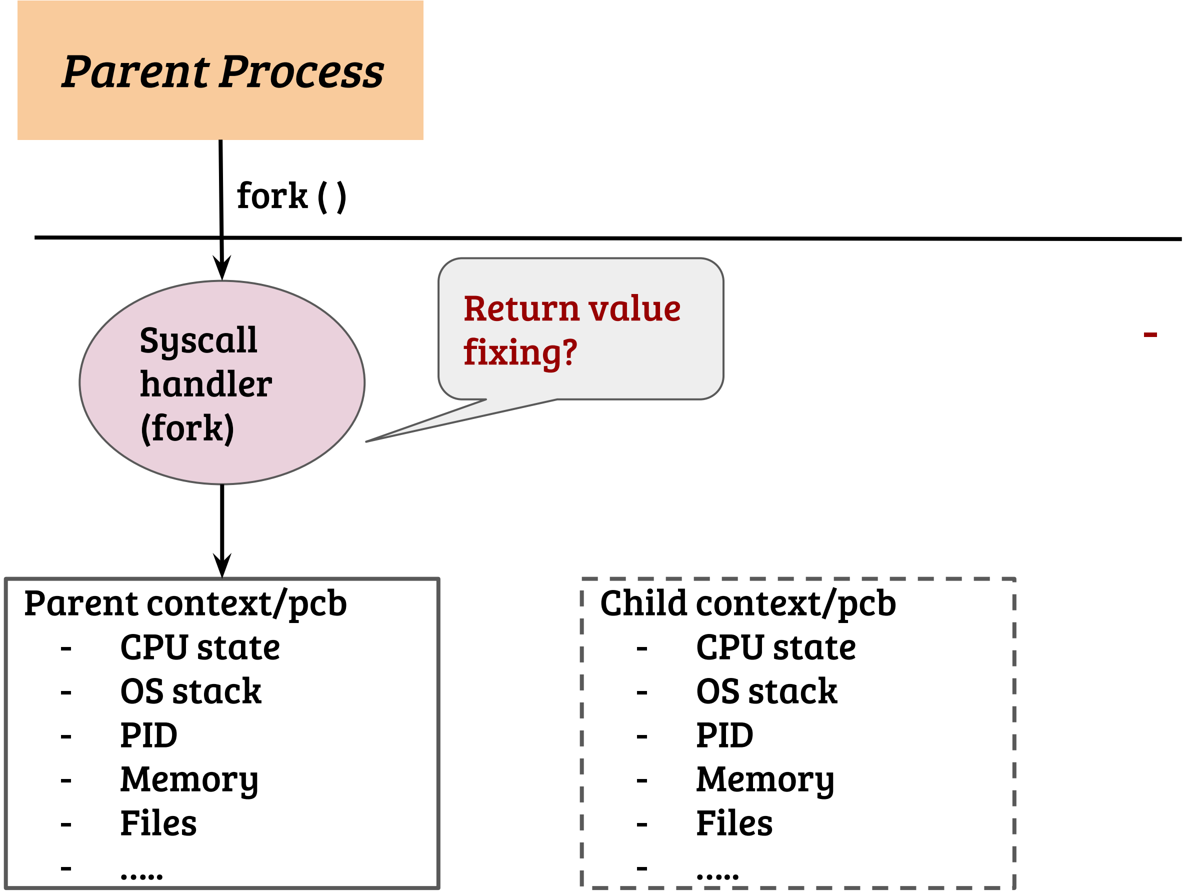


- fork( ) system call is weird -- not a typical “privileged” function call
- fork( ) creates a new process -- a *duplicate* of calling process
- On success, fork
    - Returns PID of child process to the caller (parent)
    - Returns 0 to the child

Typical implementation of fork

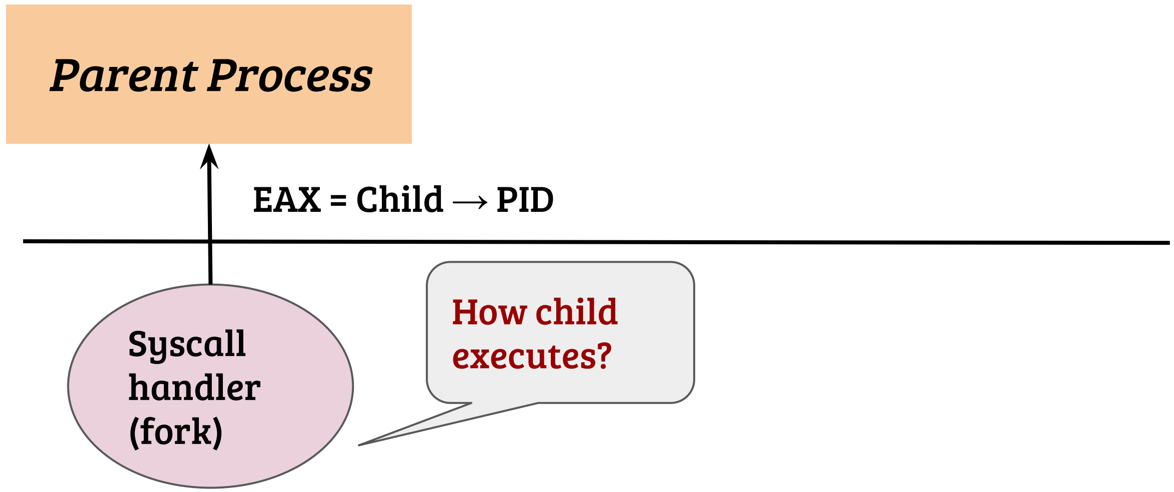


Typical implementation of fork



- Child should get ‘0’ and parent gets PID of child as return value?

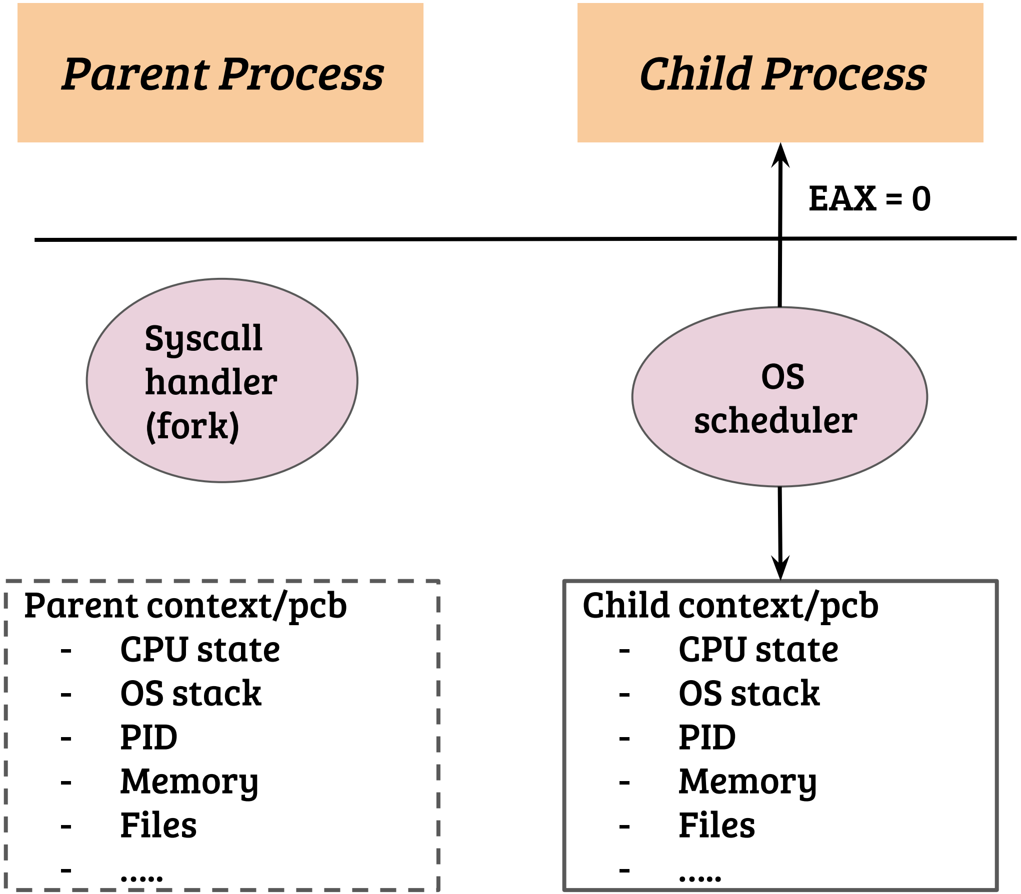
Typical implementation of fork



Parent context/pcb
- CPU state
- OS stack
- PID
- Memory
- Files
- .....

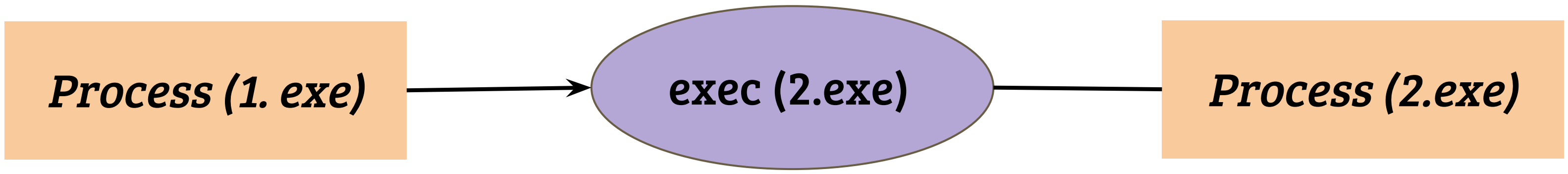
Child context/pcb
- CPU state
- OS stack
- PID
- Memory
- Files
- .....

Typical implementation of fork



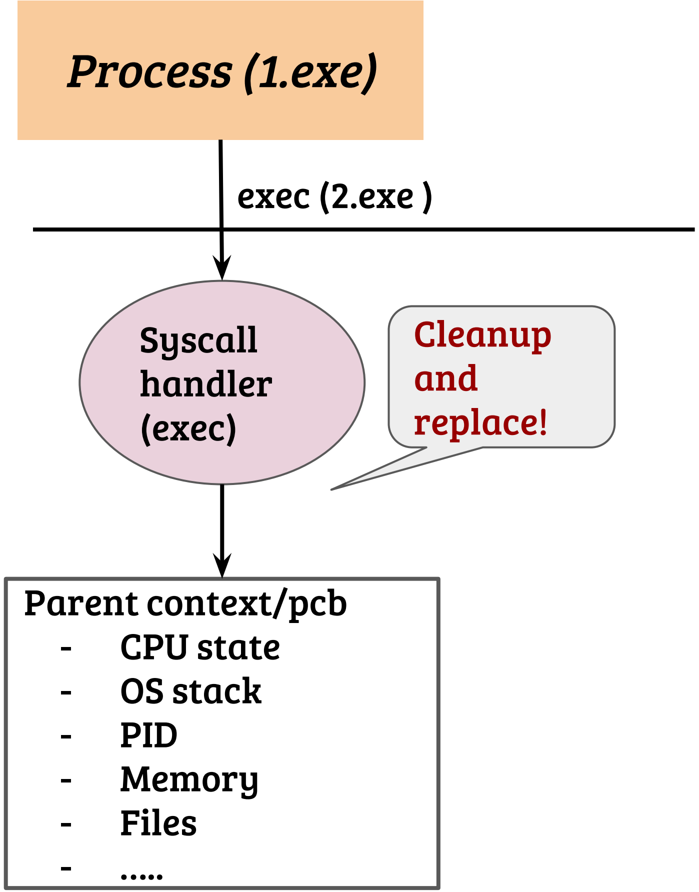
- At this point,
  - RIP is next instruction after fork( ) syscall
  - Memory is an exact copy of parent
  - Point of diversion starts from this point
- Quiz questions
  - What happens if fork( ) is called from a function (not main)?
  - What will be the stack virtual address in child?

Load a new binary - exec( )



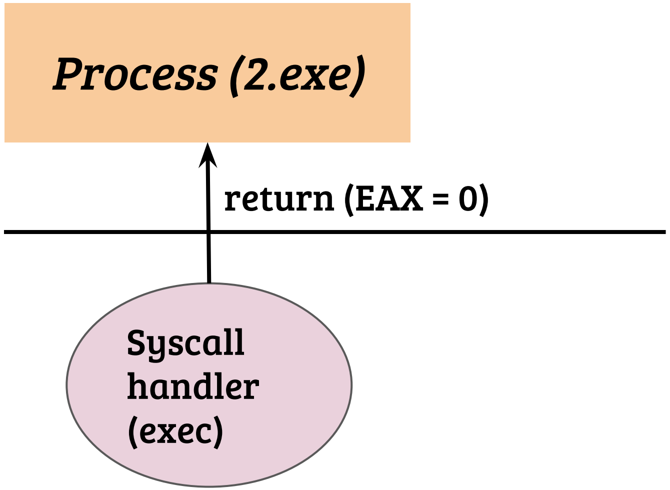
- Replace the calling process by a new executable
  - Code, data, page tables replaced by the new process
  - Usually, open files remain open (more about this latter)

Typical implementation of exec



- Remove memory mappings of calling process
- Allocate memory for code, static data and setup PT mappings
- Load the new binary (FS help required)
- Cleanup CPU state
- User RIP → address of new executable

Typical implementation of exec

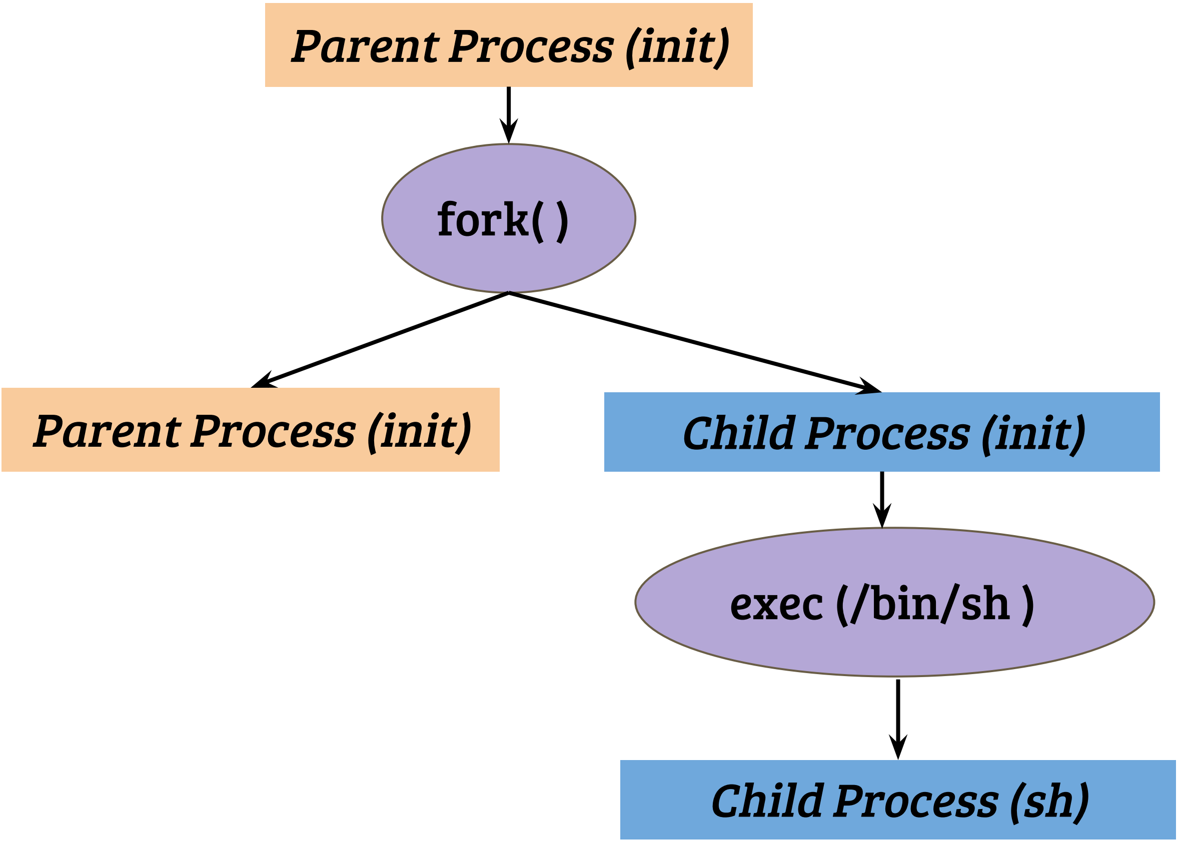


Parent context/pcb

- CPU state
- OS stack
- PID
- Memory
- Files
- .....

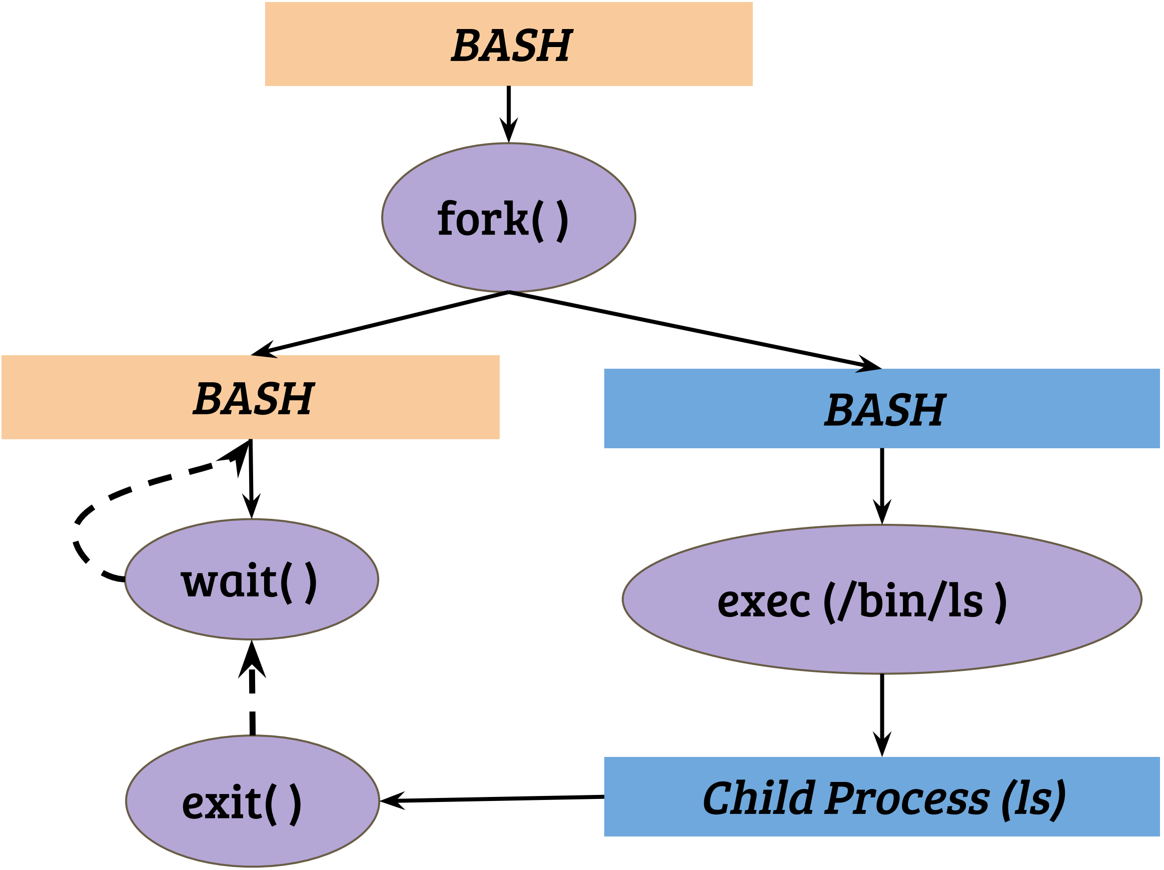
- At return, first instruction of new executable is executed
- Quiz questions
  - Will the virtual address of **main** be same as the calling process?
  - Will the physical address of **main** be same as the calling process?
  - What if the **2.exe** invokes exec with **1.exe** as parameter again?

Unix process family using fork + exec



- Fork and exec are used to create the process tree
- Commands: ps, pstree
- See the /proc directory

Shell command line: fork + exec + wait



- Parent process calls wait( ) to wait for child to finish
- When child exits, parent gets notified
- OS implementation?

Issues and optimizations

- Why copy all memory pages on fork( )? Especially, when the next system call i.e., exec( ) will wipe it out.
- Copy on write (CoW)
    - Parent and child VAs point to same PFN
    - Create a copy only when written to by parent/child
    - How?

Issues and optimizations

- Why copy all memory pages on fork( )? Especially, when the next system call i.e., exec( ) will wipe it out.
- Copy on write (CoW)
  - Parent and child VAs point to same PFN
  - Create a copy only when written to by parent/child
  - How?
- Why copy the page tables if the mappings are to be cleaned up?
  - Always schedule the child first
  - Parent suspended till the child executes exec( )
  - Example: Linux vfork ( )

Threads

- Threads are part of a process, typically execute a function
- Threads of the same process share
    - Code
    - Data
    - Files

Threads

- Threads are part of a process, typically execute a function
- Threads of the same process share
  - Code
  - Data
  - Files
- Threads have different
  - Register state
  - Stack

Threads

- Threads are part of a process, typically execute a function
- Threads of the same process share
  - Code
  - Data
  - Files
- Threads have different
  - Register state
  - Stack
- Threads can be created/destroyed faster compared to processes

Threads

- Threads are part of a process, typically execute a function
- Threads of the same process share
  - Code
  - Data
  - Files
- Threads have different
  - Register state
  - Stack
- Threads can be created/destroyed faster compared to processes
- User-level threads vs. OS level threads

Example: Linux clone( ) system call

clone(int (*fn)(void *), void *child_stack, int flags, void *arg)

- Can be used to create threads
- Flags determine what to is shared with calling process
    - CLONE_VM
    - CLONE_FILES
    - …
- In linux, fork( ), vfork( ) and pthread_create( ) invoke clone( ) system call with different combination of flags
- Why threads are lightweight?

Context switch of user space contexts

- Assume: One-to-one mapping of user threads to OS context

- What changes in the context (sw + hw)

  (1) if switching between two threads of the same process?

  (2) if switching between two processes?

  (3) if switching between two threads of different processes?

Composition of context: process vs. thread (x86)

Thread

- GPRs including SP
- CR3
- OS stack

- S/W State
  - PID
  - TID
  - VM layout
  - others

Process

- GPRs including SP
- CR3
- OS stack

- S/W State
  - PID
  - TID
  - VM layout
  - others

Switching between processes vs. threads (x86)

Thread

- GPRs including SP ✓
- CR3
- OS stack

- S/W State
  - PID
  - TID
  - VM layout
  - others

Process

- GPRs including SP ✓
- CR3
- OS stack

- S/W State
  - PID
  - TID
  - VM layout
  - others

Switching between processes vs. threads (x86)

Thread

- GPRs including SP
- CR3
- OS stack

- S/W State
  - PID
  - TID
  - VM layout
  - others

Process

- GPRs including SP
- CR3
- OS stack

- S/W State
  - PID
  - TID
  - VM layout
  - others

Switching between processes vs. threads (x86)

Thread

- GPRs including SP ✓
- CR3 ✗
- OS stack ✓

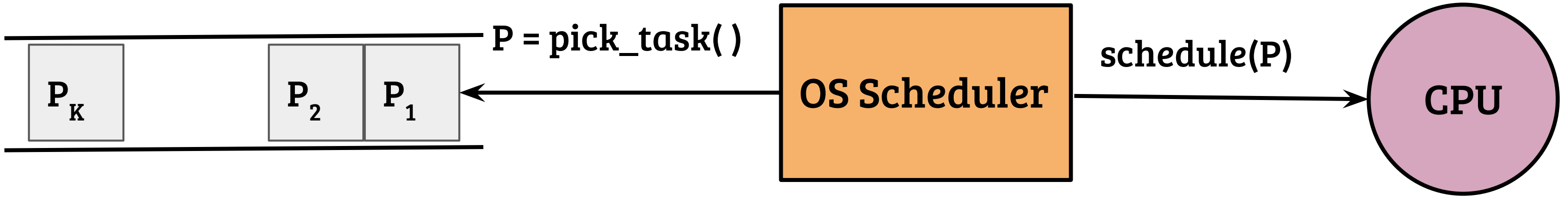
- S/W State
  - PID
  - TID
  - VM layout
  - others

Process

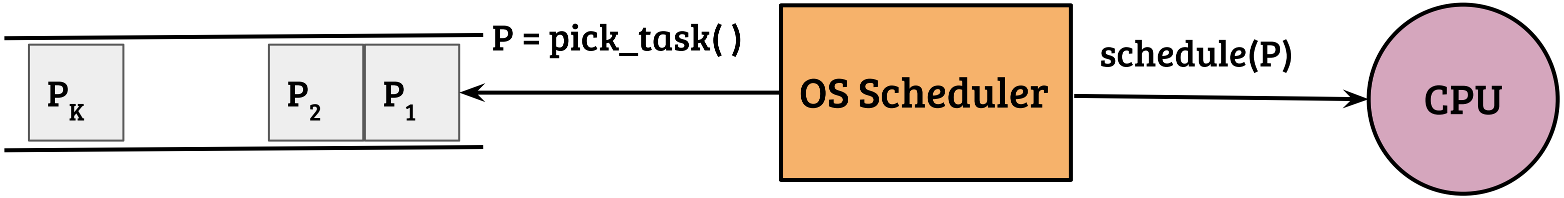
- GPRs including SP ✓
- CR3 ✓
- OS stack ✓

- S/W State
  - PID
  - TID
  - VM layout
  - others

Scheduling

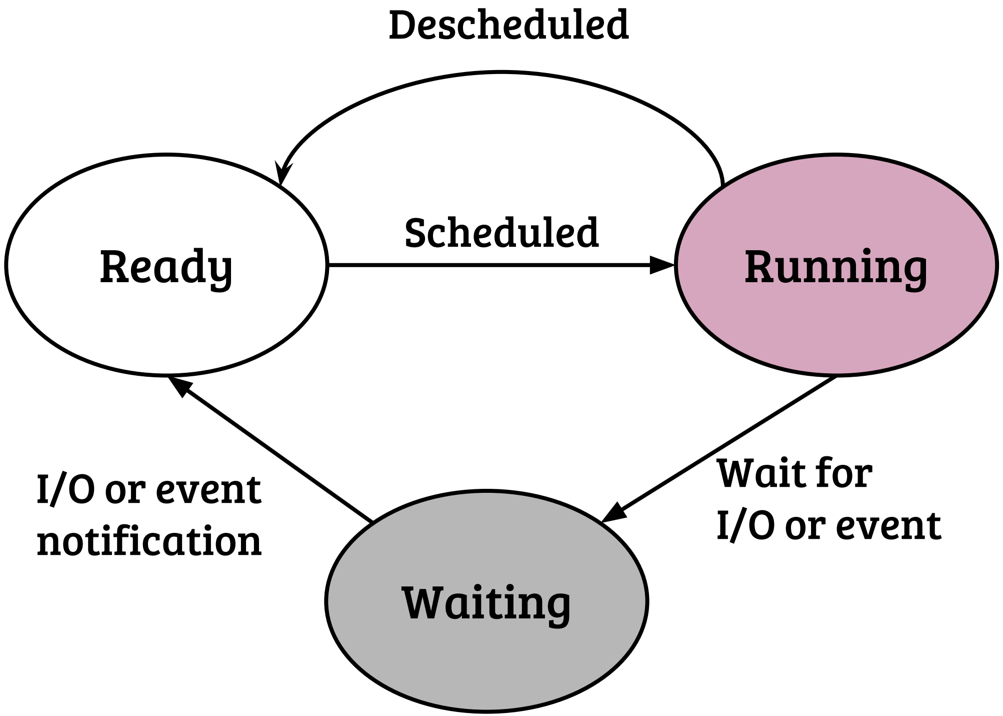


Scheduling



- Does the OS scheduler always have a choice?
- Which processes are considered? Is the list static or dynamic? How exactly, the list changes?
- When does the OS scheduler get invoked?
- What is a good scheduling strategy?

Life of a process



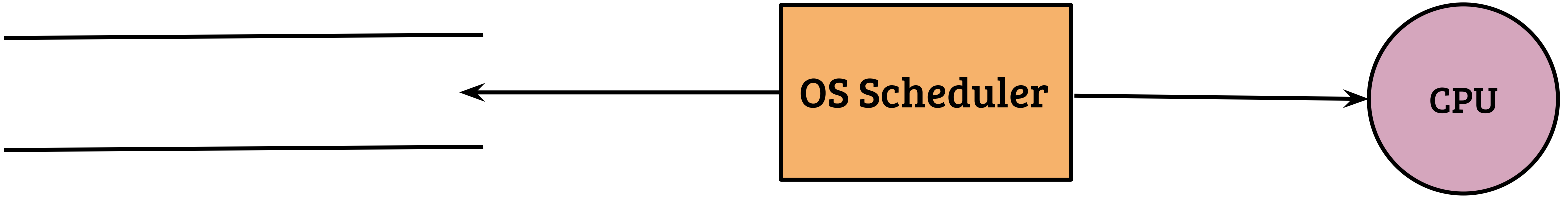
- Most processes perform a mixture of CPU and I/O activities
- Terminology: CPU burst, I/O burst, CPU bound and I/O bound
- CPU and I/O bursts during exec( ) system call?

Life of a process

Quiz: Which of the following statements are true?

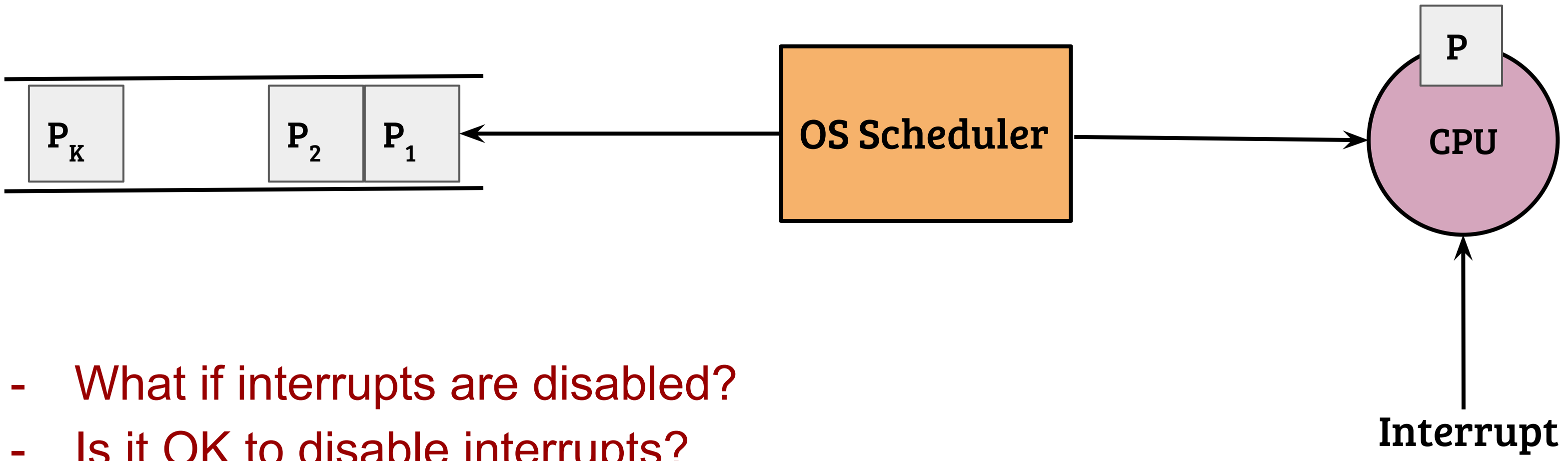
a. OS can terminate a process only in running state
b. Every process (with normal life time) is guaranteed to be in running state at least once during its life time.
c. A process can self terminate when in waiting state
d. A process can self terminate when in ready state
e. If there are N processes waiting by issuing a wait( ) system call, there must be at least N processes in running state.

Scheduling (no choice!)



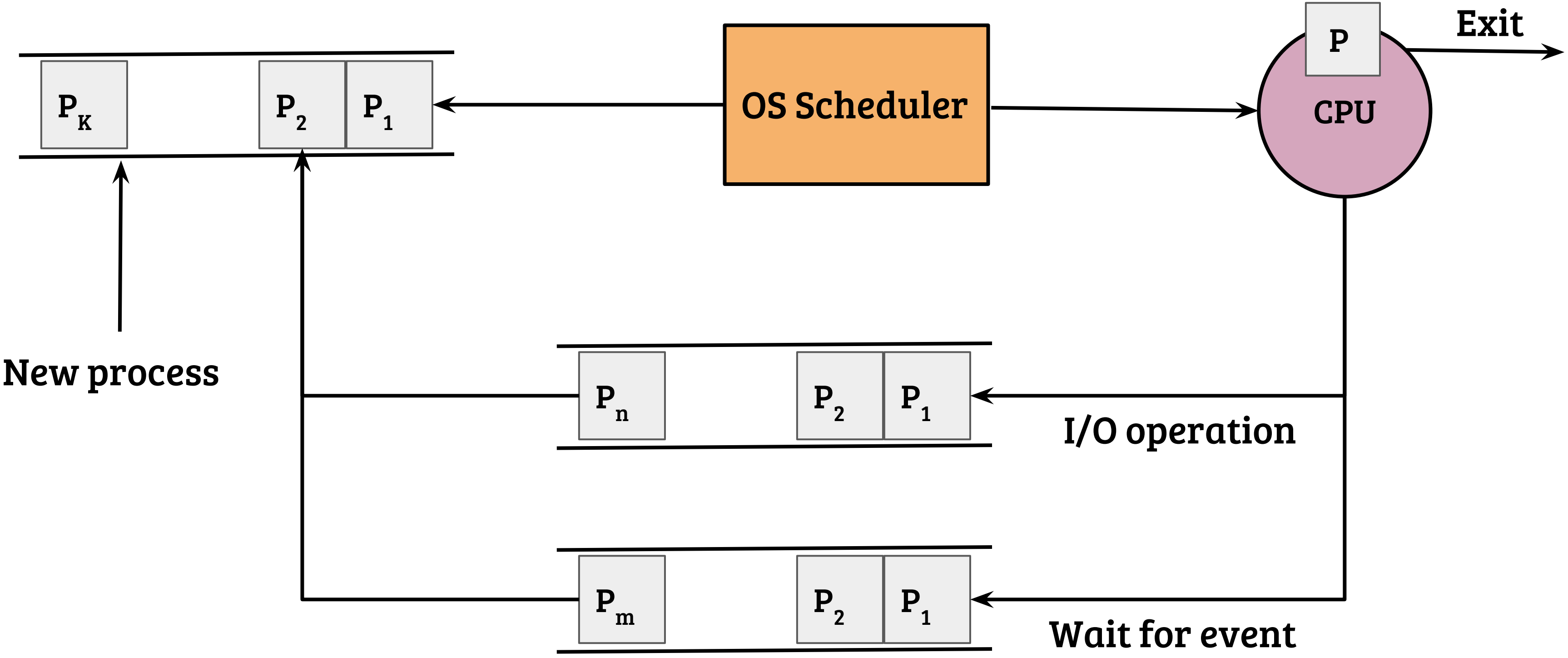
- No processes to select from!
- What should the OS scheduler do?
- A OS context is always there!
- Commonly known as the system idle process
- Can you write one for gemOS?

Scheduling (no choice!)



- What if interrupts are disabled?
- Is it OK to disable interrupts?
- Most OSs allow interrupts as much as possible, why?

Scheduler overview



Scheduler invocation

- Process creation → schedule?
- Process termination → Definitely schedule
- Process waits/blocks for an I/O or event → Definitely schedule
- CPU receives an interrupt → After handling, schedule?
- Process exclusively yields the CPU → Definitely schedule

Scheduler invocation

- Process creation → schedule?
- Process termination → Definitely schedule
- Process waits/blocks for an I/O or event → Definitely schedule
- CPU receives an interrupt → After handling, schedule?
- Process exclusively yields the CPU → Definitely schedule

- Schedulers invoked only in second, third and fifth conditions are called non-preemptive or cooperative schedulers

Preemptive scheduling

- Extent of preemption
  - Preempt when process executing in user space
  - Preempt when process is executing a system call
- Advantages
  - Interactiveness
  - Fairness (with a decent policy)
- Disadvantages
  - Scheduling overheads
  - Scheduler complexity

Scheduling strategy

- Objectives
  - Maximize throughput
  - Minimize turnaround time
  - Minimize waiting time in ready queue
  - Minimize response time
- Is the mean of above measures always enough?
- Standard deviation: fairness, predictability

Problem formulation

Process	Arrival Time	CPU bursts	I/O bursts
P1	0	0-3, 7-9, 14-15	3-7,9-14
P2	2	2-10, 12-15	10-12
P3	3	3-4, 10-11	4-10

- Every process goes through a series of CPU and I/O bursts
- Looks complicated, can it be simplified?
- What if each CPU burst is treated as a new process?