# Processes and Threads

*1DV512 – Operating Systems*

Dr. Kostiantyn Kucher

kostiantyn.kucher@lnu.se

November 10, 2020

Based on the Operating System Concepts slides by Silberschatz, Galvin, and Gagne (2018)

Suggested OSC book complement: Chapters 3 & 4

# Agenda

- Motivation and Introduction
- Processes
- Process Operations
- Threads
- Summary

# Motivation

- Typically, we expect *general-purpose* computer systems to be designed for more than a single specific program...
  - E.g., imagine a computer that can *only* run a "Hello world" program
  - Notice the focus on *general-purpose* ⇒ specialized, albeit more restricted designs can be motivated for microcontrollers, etc.
- Early computers ⇒ a single program ⇒ a single user...
- Later, the *multiprogramming* approach ⇒ batches of *jobs* (code + data) ⇒ multiple users (but they have to wait...)
- Later, the *multitasking* approach ⇒ frequently switching between multiple *tasks* ⇒ multiple users able to work interactively!
- *Process* is a more recent and more general term describing a program in *execution*

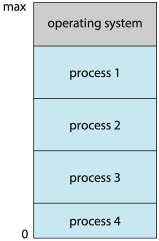


Fig. 1.12 in OSC book

# Agenda

- Motivation and Introduction
- Processes
- Process Operations
- Threads
- Summary

# Structure of a Process

- A *program* is a passive entity stored on disk (an *executable file*) ⇒ a *process* is active
- A program becomes a process when an executable file is loaded into memory (e.g., by clicking an app icon. . . )
- Same program can be launched several times (or by several users) ⇒ multiple processes!
- Process ⇒ memory contents + CPU register contents + CPU program counter state
- Typical structure of a process in memory:
  - *Text section* ⇒ executable code (fixed size!)
  - *Data section* ⇒ global variables (fixed size!)
  - *Heap* ⇒ dynamically allocated memory
  - *Stack* ⇒ temporary data / *activation records* for functions (local variables, arguments, return addresses. . . )

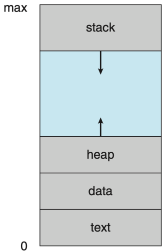


Fig. 3.1 in OSC book

# Process States

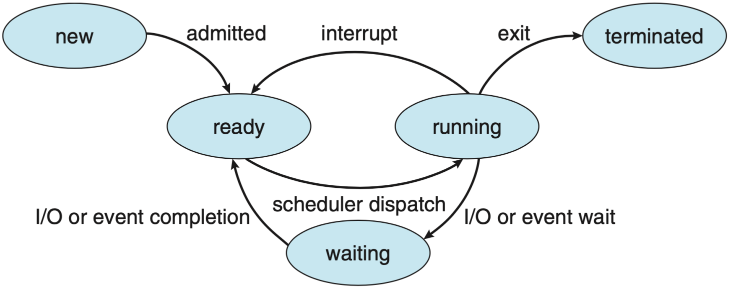


Fig. 3.2 in OSC book

As a process executes, it changes *state*, typically in the following way:

- *New* ⇒ The process is being created
- *Ready* ⇒ The process is waiting to be assigned to a processor
- *Running* ⇒ Instructions are being executed
- *Waiting* ⇒ The process is waiting for some event to occur
- *Terminated* ⇒ The process has finished execution

# Process Control Block

Each process is represented with a *process/task control block*:

- Process state ⇒ ready, running, etc.
- Process number / PID
- CPU registers ⇒ contents of all process-centric registers
- CPU scheduling information ⇒ priorities, scheduling queue pointers
- Memory-management information ⇒ memory allocated to the process
- Accounting information ⇒ CPU used, clock time elapsed since start, time limits, etc.
- I/O status information ⇒ I/O devices allocated to process, list of open files

- *Multiple* program counters *per process* ⇒ possibility of multiple control *threads*!

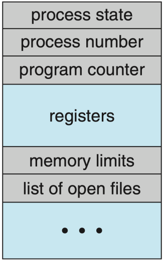


Fig. 3.3 in OSC book

# Process Scheduling

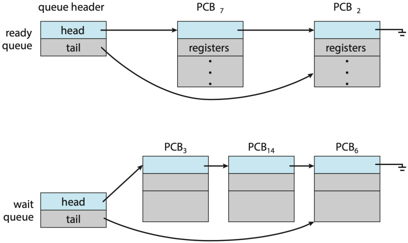


Fig. 3.4 in OSC book

- *Process scheduler* selects available processes for execution on a CPU core and maintains *scheduling queues*
- *Ready queue* $\Rightarrow$ set of all processes residing in main memory, ready, and waiting to execute
- *Wait queues* $\Rightarrow$ sets of processes waiting for particular events
- A process goes back and forth through the queues and CPU execution until it terminates
- Further details on scheduling $\Rightarrow$ next lecture!

# Queueing Diagram Representation

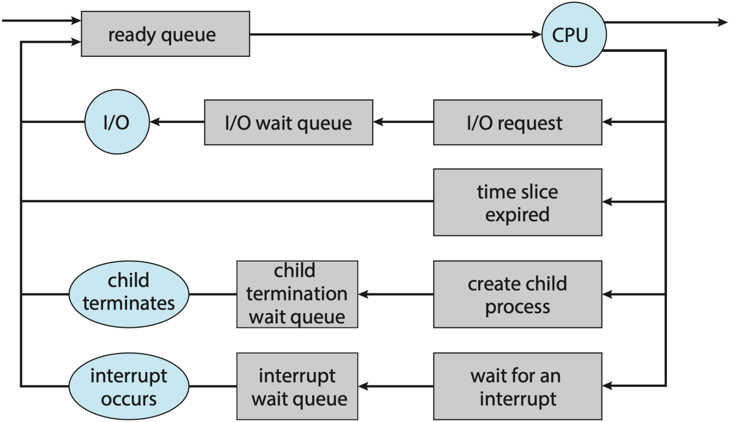


Fig. 3.5 in OSC book

# Context Switching

When switching between several processes, the *context* information must be saved
⇒ CPU register values + process state + memory management information

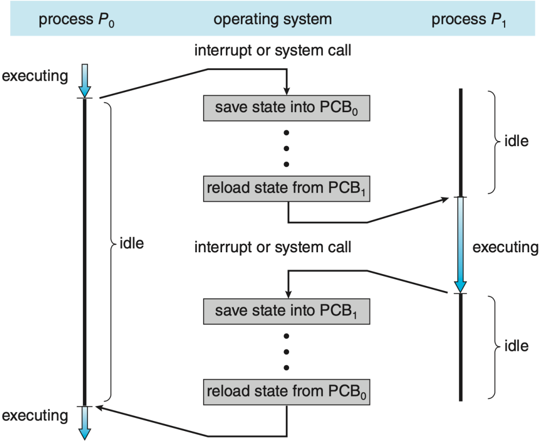


Fig. 3.6 in OSC book

# Agenda

# Typical Operations and The Process Tree

- Processes are typically identified in a unique way ⇒ *process identifier (pid)*
- OS typically support at least *creation* and *termination* of processes
- A process is typically created by another, parent process ⇒ the *process tree*
- In Unix-like systems:
  - pid 0 ⇒ scheduler (part of kernel rather than user process, not always included in the process tree)
  - pid 1 ⇒ the *init* process (in recent Linux distributions, *systemd* instead) ⇒ parent for other processes
  - Other parts of modern kernels can be exposed with pids, too, e.g., *[kthreadd]*, *[ksoftirqd]*, etc.

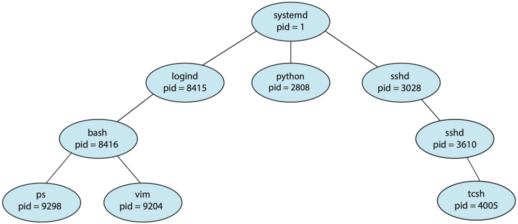


Fig. 3.7 in OSC book

# Design Choices for Process Creation

- *Resource sharing* options:
  - Parent and children share all resources
  - Children share a subset of parent's resources
  - Parent and children share no resources
- *Execution* options:
  - Parent and children execute concurrently
  - Parent waits until children terminate
- *Address space* options:
  - Child is created as a duplicate of parent
  - Child is created with a specific program loaded

# Process Creation in Unix-like Systems

- `fork()` system call creates new process ⇒ copies the memory space of the parent, and both parent and child continue to execute!
- To differentiate between such "clone" processes ⇒ use the return value of `fork()`:
  - $<0$ ⇒ error
  - $=0$ ⇒ executing within the new, child process
  - $>0$ ⇒ executing within the old, parent process ⇒ pid of the child process!
- To load and start executing a different program from disk ⇒ invoke the `exec()` system call afterwards ⇒ memory space replaced completely
- In Windows ⇒ a different program has to be specified immediately for `CreateProcess()`

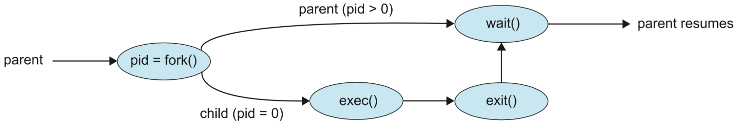


Fig. 3.9 in OSC book

# Process Termination in Unix-like Systems

- `wait()` system call $\Rightarrow$ wait (suspend execution) until *one* of the children terminates
- `waitpid()` system call $\Rightarrow$ wait (suspend execution) until any or some (= specific pid or *process group*) child terminates

- `exit()` system call (from child) $\Rightarrow$ terminate, return a status value to the parent, and release resources (memory, files, etc.)
- `abort()` system call (from parent) $\Rightarrow$ request termination of a child process

- Parent process did not invoke `wait()` yet $\Rightarrow$ the child process is a *zombie*
- Parent process terminated without invoking `wait()` $\Rightarrow$ the child process is an *orphan*

# Agenda

- Motivation and Introduction
- Processes
- Process Operations
- Threads
- Summary

# Single- and Multithreaded Processes

- *Process* ⇒ process ID, register set, stack, program counter, code section, data section, open files, ...
- *Thread* ⇒ more lightweight abstraction of a computational task ⇒ thread ID, register set, stack, program counter. ⇒ code and most data shared within the same process
- Traditional approach ⇒ a single thread of execution per process
- Later approaches ⇒ support for multithreaded processes

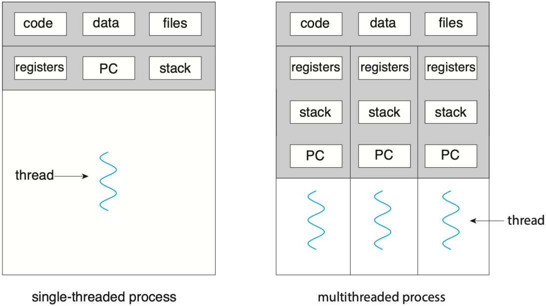


Fig. 4.1 in OSC book

# Concurrency and Parallelism

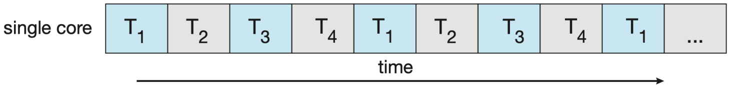


Fig. 4.3 in OSC book

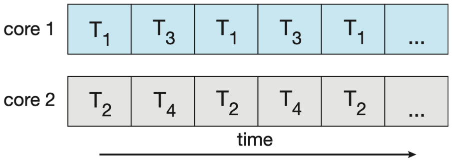


Fig. 4.4 in OSC book

# Data and Task Parallelism

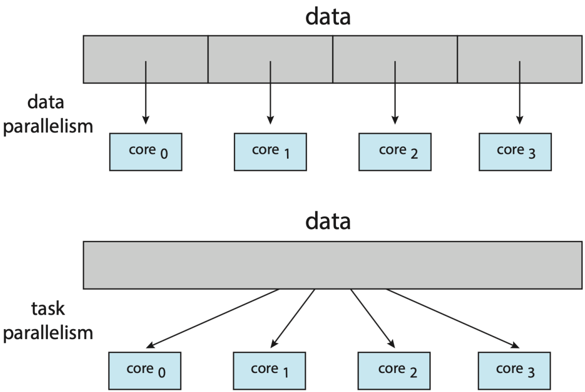


Fig. 4.5 in OSC book

# Multithreading Models

There are multiple models depending on whether the OS supports *kernel threads*, and how *user threads* are mapped to them:

- *Many-to-one* model ⇒ simple thread management, but failing to make use of multicore hardware
- *One-to-one* model ⇒ reasonable, but with overheads for creating kernel threads
- *Many-to-many* model ⇒ flexible, but difficult to implement
- *Two-level* model ⇒ adapted variation of the many-to-many model

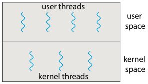


Fig. 4.6 in OSC book

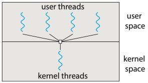


Fig. 4.7 in OSC book

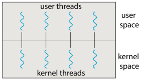


Fig. 4.8 in OSC book

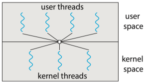


Fig. 4.9 in OSC book

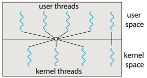


Fig. 4.10 in OSC book

# Agenda

# Summary

- Process ⇒ program in execution, including the data, resources, code, and program counter
- Based on the OS, processes change their states during their life cycles, and are typically related to each other with parent-child relationships
- Process creation paradigm in Unix-like systems ⇒ `fork() + exec()`
- Thread ⇒ more lightweight abstraction than a process, with several threads sharing the same code, data, and resources
- Multithreading can improve concurrency (and thus computational efficiency), especially for multicore hardware

- More on scheduling and process & thread synchronization issues ⇒ following lectures!