

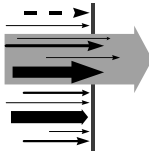
Processes



- **Process = executing program**
 - Program - sequence of instructions
 - Multiple copies of same program?
 - Program state - data context
 - Private address space?
- **Different kinds of processes?**
 - Heavyweight, lightweight, threads?

How Many Processes?

- **Single-process OS**
 - One process at a time
 - Early PC OS's
- **Multiprogramming**
 - Multiple processes
 - Most OS's



Process State

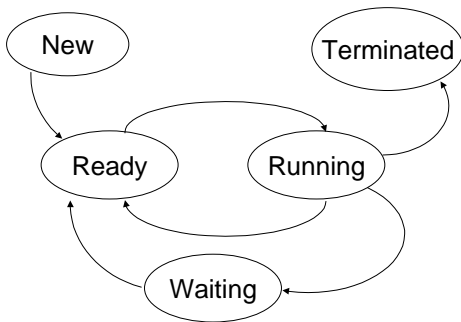
Two related meanings

- **Aggregate of all process data**
 - Registers (including PC)
 - Memory: stack, static/dynamic data
 - Memory map, I/O info
- **Process execution category**
 - New, run, ready, wait, terminated

Process States

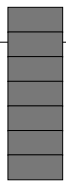
- **New** - being created
- **Running** - has control of CPU
- **Waiting** - pending some event
- **Ready** - waiting for CPU
- **Terminated** - execution complete

Process State Transitions

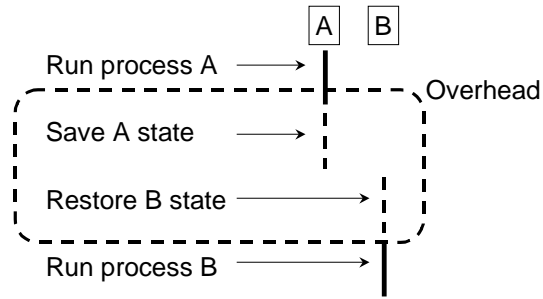


Process Control Block

- **Data structure**
 - Maintained by OS (fixed array?)
 - One PCB per process
- **Stores process info**
 - Process state, registers, mapping, I/O
 - Needed to “run” process



Context Switching



Process Scheduling

- **One running process**
 - Per processor
- **Others are waiting**
 - For processor (ready)
 - For I/O or event (wait)
 - For initialization or termination

Dispatcher M. Rosenblum - Stanford

- **Part of OS that runs processes**
 - Select process, context switch, run
- **How regain control?**
 - Trust process ("Scout's honor")?
 - System call -- process yields
 - Hardware (timer interrupt)

Dispatch Algorithm

• Which process to run next?

- First runnable in PCB array?
- Next in FIFO queue?
- Highest priority?

• If by priority, what rules?

- As set by scheduler (later)

Process Context

What gets saved?

• Registers

- PC
- Processor status
- Arithmetic
- Floating point registers and status

• Memory

- All process memory?
- Stack
- Data
- Code

Saving Memory

• Too large to store in PCB

• Write to disk?

• Leave it in place?

- Trust other processes not to disturb
- Protect with memory mapping hardware

Process Creation

- **Process origination**
 - Allocate PCB, memory
 - Load code (?)
- **Parent/child processes**
 - Present in many OS's
 - But not in all (e.g., NT)
 - A key UNIX concept!

UNIX System Startup

- **Kernel starts from boot**
- **PID 0: scheduler (swapper)**
 - System process; inside kernel
- **PID 1: init (/sbin/init)**
 - Begins multi-user state
 - Normal user process; never exits

User Process Startup

- **init creates child processes**
 - For each "tty", to handle login
- **On login, exec shell**
 - Shell process is child of init
- **User shell commands**
 - May create new child processes

Process Creation: `fork`

- **System call**
 - Executed by a process (parent)
- **Creates a duplicate process**
 - Clones memory, files, etc.
- **Call to `fork` returns in both!**
 - No guarantee which runs first

`fork` Return Status

- **How distinguish processes?**
- **Parent**
 - `fork` return value is child PID
- **Child**
 - `fork` return value is zero
 - Not a valid PID (kernel swapper PID)

Process Creation: `exec`

- **What code to run after `fork`?**
 - Initially, same code in both procs
 - Can load new code in one (child?)
- **`exec` loads new program**
 - Prior content of memory is lost
 - Can still identify parent (`getppid`)

exec Variants

- **Actually six different exec's**
 - `execl, execlp, execl`
 - `execv, execvp, execve`
- **Program pathname or PATH (p)**
- **Arguments: list (l) or argv (v)**
- **Environment: new envp (e)**

Parent and Child

- **Why create a child process?**
 - Normally to perform some action
 - E.g., execute a command for shell
 - More than one child?
- **What does parent do?**
 - Often, waits for child completion

Waiting

- **wait**
 - Waits for any child to terminate
 - Blocks until one does
- **waitpid**
 - Wait for any child or for specific PID
 - Can choose whether to block

Zombie Processes

- **Suppose child exits**
 - Can child process be destroyed?
- **Need child status for parent**
 - When (if) parent calls `wait`
- **Terminated child ⇒ zombie**
 - Until parent waits for it

Zombie Process Handling

- **What if parent exits first?**
 - All processes must have a parent
- **Orphan processes inherited**
 - Become children of `init` (PID 1)
- **When `init` child exits ...**
 - `init` calls `wait` to limit zombies
 - Sometimes known as “reaping” child

system Function

- **ANSI C library routine**
 - Invokes shell to execute command
- **Built on system calls**
 - `fork`, `exec`, `waitpid`
- **Blocks for completion**
 - No option for parent to continue
