# Creating and managing processes

<table>
<thead>
<tr>
<th>Syscall</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>fork()</strong></td>
<td>Create a child process as a clone of the current process. Return to both parent and child. Return child’s pid to parent process; return 0 to child</td>
</tr>
<tr>
<td><strong>exec</strong></td>
<td>Run application <em>prog</em> in the current process with the specified <em>args</em> (replacing any code and data that was present in process)</td>
</tr>
<tr>
<td><strong>wait</strong></td>
<td>Pause until a child process has exited</td>
</tr>
<tr>
<td><strong>exit</strong></td>
<td>Current process is complete and should be garbage collected.</td>
</tr>
<tr>
<td><strong>kill</strong></td>
<td>Send an interrupt (signal) of a specified type to a process (a bit of an overdramatic misnomer...)</td>
</tr>
</tbody>
</table>

[Unix]
Fork in action

Process 13
Program A

pid = fork();
if (pid==0)
    exec(B);
else
    wait(&status);
Fork in action

Process 13
Program A

```
pid = fork();
if (pid==0)
    exec(B);
else
    wait(&status);
```

Process 13
Program A

```
pid = fork();
if (pid==0)
    exec(B);
else
    wait(&status);
```

Process 14
Program B

```
main() {
    ...
    exec(B);
    return;
} else
    wait(&status);
```
Fork in action

main() {
...
return;
}
calls exit(0)

Process 13
Program A

pid = fork();
if (pid==0)
  exec(B);
else
  wait(&status);

Status
0

Process 14
Program B

calls exit(0)

Process 13
Program A

pid = fork();
if (pid==0)
  exec(B);
else
  wait(&status);

Process 13
Program A

PC
pid
14

PC
pid
?

PC
pid
?

PC
Fork in action

```c
#include <stdio.h>
#include <unistd.h>

int main() {

    int child_pid = fork();

    if (child_pid == 0) { // child process
        printf("I am process \%d\n", getpid());
        return 0;
    } else { // parent process
        printf("I am the parent of process \%d\n", child_pid);
        return 0;
    }
}
```

Possible outputs?
## Signals (Virtualized Interrupts)

Asynchronous notifications in user space

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Default Action</th>
<th>Corresponding Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>SIGINT</td>
<td>Terminate</td>
<td>Interrupt (e.g., CTRL-C from keyboard)</td>
</tr>
<tr>
<td>9</td>
<td>SIGKILL</td>
<td>Terminate</td>
<td>Kill program (cannot override or ignore)</td>
</tr>
<tr>
<td>14</td>
<td>SIGALRM</td>
<td>Terminate</td>
<td>Timer signal</td>
</tr>
<tr>
<td>17</td>
<td>SIGCHLD</td>
<td>Ignore</td>
<td>Child stopped or terminated</td>
</tr>
<tr>
<td>20</td>
<td>SIGSTP</td>
<td>Stop until SIGCONT</td>
<td>Stop signal from terminal (e.g., CTRL-Z from keyboard)</td>
</tr>
</tbody>
</table>
Sending a Signal

- Kernel delivers a signal to a destination process, for a variety of reasons
  - kernel detected a system event (e.g., division by zero (SIGFPE) or termination of a child (SIGCHLD) or...
  - a process invoked the kill systems call requesting kernel to send another process a signal
    - debugging
    - suspension
    - resumption
    - timer expiration
Receiving a Signal

Each signal prompts one of these default actions
- terminate the process
- ignore the signal
- terminate the process and dump core
- stop the process
- continue process if stopped

Signal can be caught by executing a user-level function called signal handler
- similar to exception handler invoked in response to an asynchronous interrupt

Process can also be suspended waiting for a signal to be caught (synchronously)
More stack magic!

- When process receives a signal
  - crosses to the kernel: current process context is copied to the process’ kernel stack
  - kernel handler copies the process context to the bottom of the signal’s stack (yet another stack!) maintained in user space
  - kernel resets context saved on the kernel stack so that PC and SP point to the user space signal handler and signal stack
  - kernel exits — and user process starts executing signal handler
  - when the handler finishes, it invokes a syscall, and the sequence is reversed
    - handler’s context copied on kernel stack; kernel handler resets original process context copying it from handler’s stack and returns from interrupt
Booting an OS

``pull oneself over a fence by one's bootstraps”

Steps in booting an O.S.:

- CPU starts at fixed address
  - in supervisor mode, with interrupts disabled
- BIOS (in ROM) loads “boot loader” code from specified storage or network device into memory and runs it
- Boot loader loads OS kernel code into memory and runs it
O.S. initialization

- Determine location/size of physical memory
- Set up initial MMU/page tables
- Initialize the interrupt vector
- Determine which devices the computer has
  - invoke device driver initialization code for each
- Initialize file system code
- Load first process from file system
- Start first process
A process is an abstraction of a running program.

The process' context captures its running state:
- registers (including PC, SP, PSW)
- memory (including the code, heap, stack)

The implementation uses two contexts:
- user context
- kernel (supervisor) context

A Process Control Block (PCB) points to both contexts and has other information about the process.
Processes can be in one of the following states:

- Initializing
- Running
- Ready (aka “runnable” on the “ready” queue)
- Waiting (aka Sleeping or Blocked)
- Zombie
More Processes than Processors

Solution: time multiplexing

- Abstractly each processor runs:
  - for ever:
    - NextProcess = scheduler()
    - Copy NextProcess->registers to registers
    - Run for a while
    - Copy registers to NextProcess->registers

- Scheduler selects some process on the ready queue
Three Flavors of Context Switching

**Interrupt:** from user to kernel space
- on system call, exception, or interrupt
- Stack switch: $P_x$ user stack $\rightarrow$ $P_x$ interrupt stack

**Yield:** between two processes, inside kernel
- from one PCB/interrupt stack to another
- Stack switch $P_x$ interrupt stack $\rightarrow$ $P_y$ interrupt stack

**Return from interrupt:** from kernel to user space
- with the homonymous instruction
- Stack switch: $P_x$ interrupt stack $\rightarrow$ $P_x$ user stack
Switching between Processes

1. Save Process 1 user registers (including SP and PC)
2. Save Process 1 kernel registers; switch SP; restore Process 2 kernel registers
3. Restore Process 2 user registers
Threads

An abstraction for concurrency
(Chapters 25-27)
Rethinking the Process Abstraction

Processes serve two key purposes:

- defines the granularity at which the OS offers isolation
  - address space identifies what can be touched by the program
- define the granularity at which the OS offers scheduling and can express concurrency
  - a stream of instructions executed sequentially
Decouple the two functionalities in two distinct abstractions:

- **A process is an abstraction of a computer**
  - its state comprises CPU, memory, devices

- **A thread is an abstraction of a core**
  - its state consists only of registers (including PC and SP)
  - it is independently schedulable by the OS
  - it lives inside some host address space
The Power of Abstractions

Infinite machines!†

Infinite cores!†

†on a single CPU (?!?)
Processes and Threads

- As previously described, processes have **one sequential thread of execution**

- Many OSs offer the ability to have **multiple concurrent threads execute in a process**
  - Individual threads perform one instruction at a time
  - Multiple threads in a process allow multiple tasks to be performed concurrently, at the same time (at least, logically)

- Resources are managed differently:
  - **CPU state managed on a per-thread basis**
  - **All other resources on a per-process basis**
Why Threads?

- **To express a natural program structure**
  - updating the screen, fetching new data, receiving user input
    - different tasks within the same address space

- **To exploit multiple processors**
  - different threads may be mapped to distinct processors

- **To maintain responsiveness**
  - slow, long running task performed by background threads
  - foreground threads respond immediately to user interactions

- **Masking long I/O device latency in blocking syscalls**
  - do useful work while waiting
Multithreading: Responsiveness

- Common web browser pattern:
  - UI thread draws web page, handles mouse clicks
  - Pool of background threads downloads web pages from remote web servers

- Does this require multiple CPUs to yield a benefit?
  - NO!
  - BG threads will usually be blocked on I/O
  - Ditto for UI thread

- Even with a single processor, multithreading can greatly improve application responsiveness
  - especially when tasks are I/O bound
Multithreading: Scalability

- A large scientific/mathematical computation:
  - instead of using a single thread, split in multiple concurrently executing threads

- Does this require multiple CPUs to yield a benefit?
  - YES!
  - Threads will be mostly CPU bound, not I/O bound
  - With only one CPU, multithreading will actually likely slow execution, not speed it up!
    - (context switches, synchronization overheads, etc)

- On the other hand... A single-threaded process cannot take advantage of multiple CPUs
  - need either multiple processes, or one process with multiple threads
Multithreaded Processing Paradigms

Dispatcher/Workers

Specialists

Pipeline
Implementing threads

Shared State
- Heap
- Global Variables
- Code

Per-Thread State
- Thread Control Block (TCB)
  - Stack pointer
  - Other Registers (PC, etc)
  - Thread metadata (ID, priority, etc)
- Stack
  - Stack frame
  - Stack frame
  - Stack frame

Per-Thread State
- Thread Control Block (TCB)
  - Stack pointer
  - Other Registers (PC, etc)
  - Thread metadata (ID, priority, etc)
- Stack
  - Stack frame

Note: No protection enforced at the thread level!
Processes vs. Threads: Parallel lives

Processes

- Have data/code/heap and other segments
- Include at least one thread
- If a process dies, its resources are reclaimed and its threads die
- Interprocess communication via OS and data copying
- Have own address space, isolated from other processes
- Each process can run on a different processor
- Expensive creation and context switch

- No data segment or heap
- Needs to live in a process
- More than one can be in a process. First calls main.
- If a thread dies, its stack is reclaimed
- Inter-thread communication via memory
- Have own stack and registers, but no isolation from other threads in the same process
- Each thread can run on a different processor
- Inexpensive creation and context switch