Processes

Demo: Virtualizing the CPU

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <assert.h>
#include "common.h"

int
main(int argc, char *argv[])
{
    if (argc != 2) {
        fprintf(stderr, "usage: cpu <string>
        exit(1);
    }
    char *str = argv[1];
    while (1) {
        Spin(1);
        printf("%s\n", str);
    }
    return 0;
}
```

```
prompt> gcc -o cpu cpu.c -Wall
prompt> ./cpu "A"
A
A
A
A
^C
prompt>
```

```
prompt> ./cpu A & ./cpu B & ./cpu C & ./cpu D &
[1] 7353
[2] 7354
[3] 7355
[4] 7356
A
B
D
C
A
B
D
C
A
```

the OS makes each application think it has the system resources (CPU, memory) to itself

the CPU abstraction is the *process*

Processes

- a *process* is a running program
  - but programs and processes are distinct – multiple copies of the same program can be running simultaneously, each as a separate process
- the *machine state* of a process includes
  - memory
    - the *address space* of the process is the memory it can access
    - holds code+data from the executable, stack and heap for runtime use
  - execution context of the process
    - values stored in registers – PC and process data
  - system resources allocated to the process
    - e.g. open file descriptors
  - security attributes
    - e.g. its owner and its set of permissions
- switching from one process to another requires a *context switch*

Processes

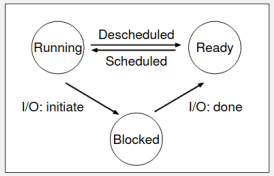
- mechanisms vs policies
  - low-level *mechanisms* support the implementation of functionality
  - high-level *policies* are algorithms for making decisions

mechanism	vs	policy
context switch		which process should run now

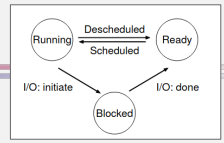
Process State

Three main states –

- *running* – the process is running on the CPU (instructions are being executed)
- *ready* – the process is ready to run but is not currently running
- *blocked* – the process is not ready to run (it is waiting for an I/O request to complete)



Process State Transition



Time	$Process_0$	$Process_1$	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	
4	Running	Ready	$Process_0$ now done
5	–	Running	
6	–	Running	
7	–	Running	
8	–	Running	$Process_1$ now done

- two processes, no I/O
  - both are always ready to run

Time	$Process_0$	$Process_1$	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	$Process_0$ initiates I/O
4	Blocked	Running	$Process_0$ is blocked,
5	Blocked	Running	so $Process_1$ runs
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	$Process_1$ now done
9	Running	–	
10	Running	–	$Process_0$ now done

- two processes, one does I/O

Linux Processes

- viewing processes
  - `ps`
  - `top`

```
By default, ps selects all processes with the same effective user ID
(euid=EUID) as the current user and associated with the same terminal
as the invoker.  It displays the process ID (pid=PID), the terminal
associated with the process (tname=TTY), the cumulated CPU time in
[DD-]hh:mm:ss format (time=TIME), and the executable name (ucmd=CMD).
Output is unsorted by default.
```

```
To see every process on the system using BSD syntax:
   ps ax
   ps axu
```

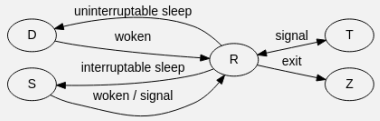
a – lift the "only yourself" restriction (show processes from all users)
x – lift the "must have a tty" restriction (show all processes, not just the ones associated with this terminal)
u – display user-oriented format

- killing processes
  - `kill`, k (in `top`)
- cpu info
  - `cat /proc/cpuinfo`
  - `nproc`

Linux Processes

- process states and state transitions

```
D    uninterruptible sleep (usually IO)
I    Idle kernel thread
R    running or runnable (on run queue)
S    interruptible sleep (waiting for an event to complete)
T    stopped by job control signal
t    stopped by debugger during the tracing
W    paging (not valid since the 2.6.xx kernel)
X    dead (should never be seen)
Z    defunct ("zombie") process, terminated but not reaped by its parent
```



Process Control Block (PCB)

- the *process control block* is a kernel data structure storing all of the info about a process
  - process identifier (PID)
  - process state (running, ready, blocked, terminated, …)
  - pointers to other related processes (parent, children)
  - saved CPU context (registers) of process when it is not running
  - information related to memory locations of a process
  - information related to ongoing I/O communication
  - …
- known as `task_struct` in Linux

Process API

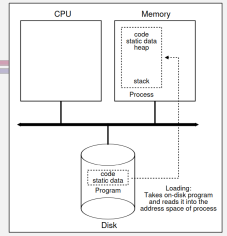
- **Create:** An operating system must include some method to create new processes. When you type a command into the shell, or double-click on an application icon, the OS is invoked to create a new process to run the program you have indicated.
- **Destroy:** As there is an interface for process creation, systems also provide an interface to destroy processes forcefully. Of course, many processes will run and just exit by themselves when complete; when they don't, however, the user may wish to kill them, and thus an interface to halt a runaway process is quite useful.
- **Wait:** Sometimes it is useful to wait for a process to stop running; thus some kind of waiting interface is often provided.
- **Miscellaneous Control:** Other than killing or waiting for a process, there are sometimes other controls that are possible. For example, most operating systems provide some kind of method to suspend a process (stop it from running for a while) and then resume it (continue it running).
- **Status:** There are usually interfaces to get some status information about a process as well, such as how long it has run for, or what state it is in.

running a program

`kill` signals
- `SIGKILL`
- `SIGSTOP`
- `SIGCONT`
- `SIGINT`

`ps`, `top`, etc

Process Creation



Steps –

- load the program code and any static data into memory (in the address space of the process)
  - reads from disk
- allocate memory for the stack and initialize it with the parameters for main
  - includes memory for parameters, declared variables (local variables), return value, return address
- allocate memory for the heap
  - heap is for dynamically allocated memory
- set up open file descriptors
  - `stdin`, `stdout`, `stderr`