Motivation

• On UNIX, each computing task is represented by a \textit{process}.
• UNIX runs many tasks \textit{seemingly at the same time}
• Each process receives a little slice of CPU time by the \textit{scheduler}.
What is a process?

• A process is something of a container, bundling:
  • a running application.
  • Process id: index into the system process table.
  • It’s own address space.
  • its environment variables.
  • Open files/resources.
• Most processes come and go rapidly, as tasks start and complete

• So, where does the **first process** come from?

• On UNIX, some processes run **from system boot to shutdown**

• The kernel spawns the **first process** during the boot sequence

• The first process is *init* and its PID is 1.
How many processes do we have?

- UNIX system has a **finite**, yet large **pool of processes**
- In practice, a system **almost never runs out** of processes
- Each new task -- say, launching pico -- immediately allocates a process from the pool with a **unique PID**

```
$ sleep 10 & ps -o pid,command,state,stime

<table>
<thead>
<tr>
<th>PID</th>
<th>COMMAND</th>
<th>S</th>
<th>STIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>16351</td>
<td>-bash</td>
<td>S</td>
<td>11:23</td>
</tr>
<tr>
<td>16845</td>
<td>sleep 10</td>
<td>D</td>
<td>11:42</td>
</tr>
<tr>
<td>16846</td>
<td>ps -o pid,u</td>
<td>R</td>
<td>11:44</td>
</tr>
</tbody>
</table>
```
Forking a new process

• In UNIX, the `fork()` system call is used to spawn a new process
• A “child” process is a clone of the “parent” process:
  • Complete copy of memory image.
  • All open files
• Each process has its own pid.

• Have you ever “fork”-ed a process?
You are always forking new processes

• The `ls` command in the shell prompt is actually a “child” process
• Who is the parent process?

• If a user types the `ls` command in a shell, a new process is forked.
• The **Linux kernel** duplicates the shell's pages of memory, and executes the `ls` command.
**fork()**

- The *fork()* system call returns twice: in the **parent** process and in the **child** process.
- In the “**parent**” process it returns the PID of the “**child**” process.
- While in the “**child**” process it returns 0.
- -1 is returned if fork fails.

```c
#include <sys/wait.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>

int main(int argc, char *argv[]) {
    pid_t cpid;
    int status;

    cpid = fork();
    if (cpid == -1) {
        perror("fork"); exit(EXIT_FAILURE);
    }
    if (cpid == 0) { /* Code executed by child */
        printf("PID is %ld\n", (long) getpid());
        exit(argc==2?atoi(argv[1]):0);
    }
    else { /* Code executed by parent */
        printf("Child PID is %ld\n", (long)cpid);
        wait(&status); /* waits on the exit status from the child*/
        if (WIFEXITED(status))
            printf("%ld exit status = %d\n", (long)cpid,
                    WEXITSTATUS(status));
        printf("%ld exited with status %ld\n", (long)cpid,(long)status);
        exit(0);
    }
}
Parent and Child - Example

$>./fork_example 2
Child PID is 29894
PID is 29894
exit status = 2
29894 exited with status 512

$>./fork_example 1 & ps --user jumanan -o pid,ppid,command,stime,time
[1] 29958
Child PID is 29960
PID is 29960

<table>
<thead>
<tr>
<th>PID</th>
<th>PPID</th>
<th>COMMAND</th>
<th>STIME</th>
<th>TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>29345</td>
<td>29344</td>
<td>-tcsh</td>
<td>14:39</td>
<td>00:00:00</td>
</tr>
<tr>
<td>29958</td>
<td>29345</td>
<td>./fork_example 1</td>
<td>15:10</td>
<td>00:00:00</td>
</tr>
<tr>
<td>29959</td>
<td>29345</td>
<td>ps --user jumanan -o pid,pp</td>
<td>15:10</td>
<td>00:00:00</td>
</tr>
<tr>
<td>29960</td>
<td>29958</td>
<td>./fork_example 1 S</td>
<td>15:10</td>
<td>00:00:00</td>
</tr>
</tbody>
</table>

$> 29960 exit status = 1
29960 exited with status 256

[1] Done

$./fork_example 1
Process Z has the same environment variables as A, the same memory content, the same program state, and the same files/resources open
Copy-On-Write

- “Parent” and “child” processes run "simultaneously"

- They use the **same resources** until one of them decides to **change** the data

- Then, a unique copy of the considered data is duplicated for its use (**copy-on-write**)
execvp()

- **After the fork**, *Process A* might continue running the same application.

- However, *Process Z* might immediately choose to **execute another application**, which is done using the *execvp()* system call.

- It **loads** a new program, and **overrides** the parent’s one

- **int execvp(const char *file, char *const argv[]);**
exec() – (cont.)

```c
int runcmd() {
    char* argv[MAX_ARGS] = {"ls","-all",0}; /* have to end with */
    int child_status;
    pid_t child_pid;
    child_pid = fork();
    if(child_pid == 0) {
        /* This is done by the child process. */
        execvp(argv[0], argv);
        /* If execvp returns, it must have failed. */
        printf("Unknown command\n");
        exit(0);
    } else {
        /* This is run by the parent. Wait for the child to terminate. */
        pid_t tpid = wait(&child_status);
        return child_status;
    }
}
```
Inter Process Communication

• Mechanism for processes to communicate and to synchronize their actions
• If \( P \) and \( Q \) wish to communicate, they need to:
  • establish a communication link between them
  • exchange messages via send/receive
• Implementation of communication link
  • physical (e.g., shared memory, hardware bus)
  • logical (e.g., logical properties: FIFO)
• The logical communication link could be:
  • Direct - Signals
  • Indirect - Pipes
Signals – Direct Communication

• The source process can "raise" a signal and have it delivered to destination process.
• The destination process' signal handler is invoked and the process can handle it.
• A direct communication in which unidirectional channels are established automatically.
• Processes must name each other explicitly using the process ID in order to send messages of fixed size.
• Asynchronous.

What types of signals you are familiar with?

More info read: http://www.tldp.org/LDP/SLK/ipc/ipc.html
PIVES – Indirect communication

• A direct communication in which **uni-directional channels** are established between “related” processes

• Basically, a call to the `int pipe(int fd[2])` system call attaches a pair of file descriptors to the pipe

• One of these descriptors is connected to the **write end** of the pipe, and the other is connected to the **read end**

• On many systems, pipes will fill up after you write about 10KB to them without reading anything out

```c
write()  fd[1]  PIPE  fd[0]  read()
```
Simple example

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

char *msg1 = "hello #1";
char *msg2 = "hello #2";
char *msg3 = "hello #3";

main()
{
    char inbuf[MSGSIZE];
    int p[2], j;
    /* open pipe */
    if(pipe(p) == -1)
    {
        perror("pipe call error");
        exit(1);
    }
    /* write down pipe */
    write(p[1], msg1, MSGSIZE);
    write(p[1], msg2, MSGSIZE);
    write(p[1], msg3, MSGSIZE);
    /* read pipe */
    for(j=0; j<3; j++)
    {
        read(p[0], inbuf, MSGSIZE);
        printf("%s\n", inbuf);
    }
    exit(0);
}
```
Example – “Parent” and “Child”

```
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

int main(int argc, char *argv[]) {
    int pipefd[2];
    pid_t cpid;
    char buf;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s <string>\n", argv[0]);
        exit(EXIT_FAILURE);
    } if (pipe(pipefd) == -1) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }
    cpid = fork();
    if (cpid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }
    if (cpid == 0) { /* Child reads from pipe */
        close(pipefd[1]); /* Close unused write end */
        while (read(pipefd[0], &buf, 1) > 0)
            write(STDOUT_FILENO, &buf, 1);
        write(STDOUT_FILENO, "\n", 1);
        close(pipefd[0]);
        _exit(EXIT_SUCCESS);
    } else { /* Parent writes argv[1] to pipe */
        close(pipefd[0]); /* Close unused read end */
        write(pipefd[1], argv[1], strlen(argv[1]));
        close(pipefd[1]); /* Reader will see EOF */
        wait(NULL); /* Wait for child */
        exit(EXIT_SUCCESS);
    }
}
```