CPSC 457

Processes - part 1

Contains slides from Mea Wang, Andrew Tanenbaum and Herbert Bos

Process

- we want the ability to run multiple programs at the same time (multitasking)
- for this we need firm control and compartmentalization of the various programs
- to this end we create an abstraction of **program in execution** and call it a **process**

Multitasking

- we need the process abstraction to implement **multitasking**
- multitasking allows an illusion of parallelism
  - running **N** processes with **M** CPUs, while **N** > **M**
  - works even with a single CPU:

```
run program (i) for a fraction of a second
switch to program (i+1)
repeat
```

- multitasking allows us to reduce CPU idling during I/O
  - CPU could be given to another process rather than remain idle
- multitasking is only practical when memory is big enough to hold multiple running programs

Program != Process

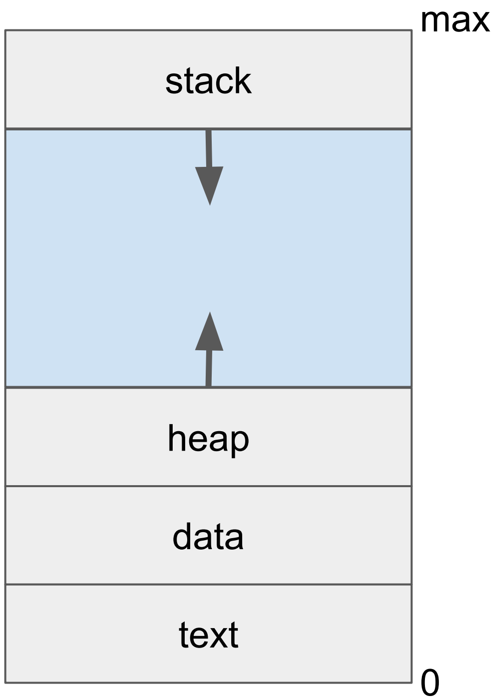
- a program is a **passive** entity
  - an executable file containing a list of instructions, usually stored on disk
- a process is an **active** entity
  - associated with a program counter and other resources
- a program **becomes** a process when it is loaded into memory for execution
- a single program can be used to create multiple processes
  - eg. running multiple instances of one executable

Analogy: baking and OS

Baking		OS
recipe	○	program
the chef	○	CPU
flour, eggs, sugar	○	input data
reading the recipe, getting the ingredients, baking the cake	○	process
phone call	○	interrupt

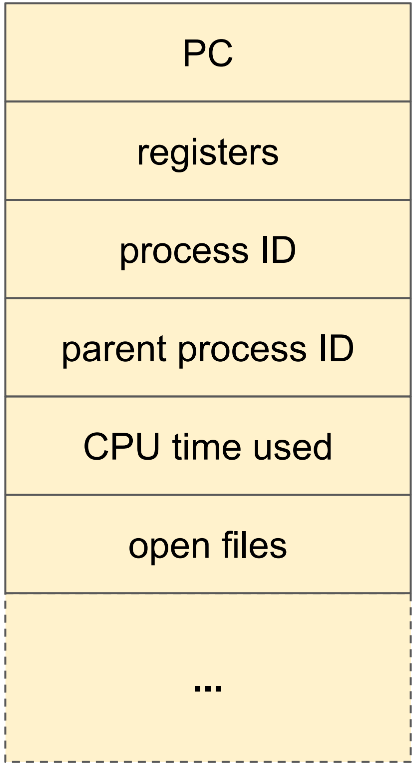
A process in memory

- each process gets its own **address space**
  - part of memory available to a process, decided by OS
  - on modern OSes it is a *virtual* address space (0 - max), isolated from other processes
- **text section**: the program code
- **data section:** global variables, constant variables
- **heap**: memory for dynamic allocation during runtime
- **stack**: temporary data (parameters, return address, local variables)
- plus many other bits of information needed by the OS for management, usually grouped in a Process Control Block data structure



Process control block (PCB)

- each process is represented in the OS by a PCB that includes:
  - process state
  - program counter
  - CPU registers
  - CPU-scheduling info priority, pointers to the queue, other parameters
  - memory management info: page tables, segment tables, etc.
  - accounting info: CPU time, timeout values, process numbers, etc.
  - I/O status info: open files, I/O devices, etc.
- the **process table** is a collection of all PCBs



Some of the fields of a PCB

Process management	Memory management	File management
program counter registers stack pointer process state priority scheduling parameters process ID parent process process group signals process start time CPU time used children's CPU time used time of next alarm ...	pointer to text segment pointer to data segment pointer to stack segment ...	root directory working directory file descriptors user ID group ID ...

Look for "`task_struct`" in Linux kernel sources

https://github.com/torvalds/linux/blob/master/include/linux/sched.h

Operations on processes

- processes need to be created and deleted dynamically and OS must provide mechanisms for this
- process creation, eg. **`fork()`** in UNIX
    - **parent process** - the process that is creating a new process
    - **child process** (**child**) - the newly created process
    - processes in the system form a **process tree**
    - each process gets **PID** - a unique **process identifier**
- process execution, eg. **`fork()`**
- process termination, eg. **`exit()`** or **`kill()`**
    - to let the OS delete the process
    - termination can be (typically) only requested by the process or its parent
- other operations: synchronization, communication, ...

A multiprocess program in C

```
$ man fork

pid_t fork(void);

fork()  creates  a new process by duplicating the calling process.  The
new process is referred to as the child process.  The  calling  process
is referred to as the parent process.

The child process and the parent process run in separate memory spaces.
At the time of fork() both memory spaces have the same content.  Memory
writes,  file  mappings (mmap(2)), and unmappings (munmap(2)) performed
by one of the processes do not affect the other.

The child process is an exact duplicate of the  parent  process  except
for the following points:
...
```

A multiprocess program in C

```c
#include <unistd.h>

int main()
{
    printf("Hello\n");
    /* create & run child process - a duplicate of parent */
    fork();
    /* both parent and child will execute the next line */
    printf("world.\n");
}
```

```
Possible output:


???
```

A multiprocess program in C

```
#include <unistd.h>

int main()
{
    printf("Hello\n");
    /* create & run child process - a duplicate of parent */
    fork();
    /* both parent and child will execute the next line */
    printf("world.\n");
}
```

```
Possible output:

Hello
world.
world.
```

Are other outputs possible?

A multiprocess program in C


```c
#include <unistd.h>

int main()
{
    printf("Hello\n");
    /* create & run child process - a duplicate of parent */
    fork();
    /* both parent and child will execute the next line */
    printf("world.\n");
}
```

```
Possible output:

Hello
world.
world.
```

```
Another possible output:

Hello
worwold.
rld.
```

Are other outputs possible?

A multiprocess program in C


```c
#include <unistd.h>

int main()
{
    printf("Hello\n");
    /* create & run child process - a duplicate of parent */
    fork();
    /* both parent and child will execute the next line */
    printf("world.\n");
}
```

If fork() fails, it returns **-1**. You should always check the return value of a system call.

```
Possible output:

Hello
world.
world.
```

```
Another possible output:

Hello
worwold.
rld.
```

```
Output if fork() fails:

Hello
world.
```

A multiprocess program in C


```c
int main()
{
    /* create & run child process - a duplicate of parent
     * and remember the return value */
    pid_t  pid = fork();
    /* both parent and child will execute the next line,
     * but will have different value for pid:
     * 0 for child, non-zero for parent */
    printf("My pid is %d.\n", pid);
}
```

```
Possible output:

My pid is 7.
My pid is 0.
```

```
Possible output:

My pid is 0.
My pid is 7.
```

```
Possible output:

My pid is -1.
```

Homework - can you predict the output?

```
int main()
{
    fprintf( stderr, "A\n");
    fork();
    fprintf( stderr, "B\n");
    fork();
    fprintf( stderr, "C\n");
}
```

```
int main()
{
    for(int i=0 ; i<4 ; i++ ) {
        fork();
    }
    printf("X");
}
```

Fork bomb

```
int main()
{
    while(1) {
        fork();
    }
    printf("X");
}
```

WARNING: please do not run this on CPSC servers.

A multiprocess program in C

```
$ man execl

int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
...


The  exec() family of functions replaces the current process image with
a new process image.  The functions described in this manual  page  are
front-ends  for execve(2).  (See the manual page for execve(2) for fur-
ther details about the replacement of the current process image.)

The initial argument for these functions is the name of a file that  is
to be executed.

...
```

A multiprocess program in C

```
$ man execve

int execve(const char *filename, char *const argv[], char *const envp[]);

execve() executes the program pointed to by filename.  filename must be
either a binary executable, or a script starting with  a  line  of  the
form:
             #! interpreter [optional-arg]

For details of the latter case, see "Interpreter scripts" below.

argv is  an  array  of argument strings passed to the new program.  By
convention, the first of these  strings  should  contain  the  filename
associated  with the file being executed.  envp is an array of strings,
conventionally of the form key=value, which are passed  as  environment
to  the  new  program.  Both argv and envp must be terminated by a null
pointer.
...


...
```

A multiprocess program in C

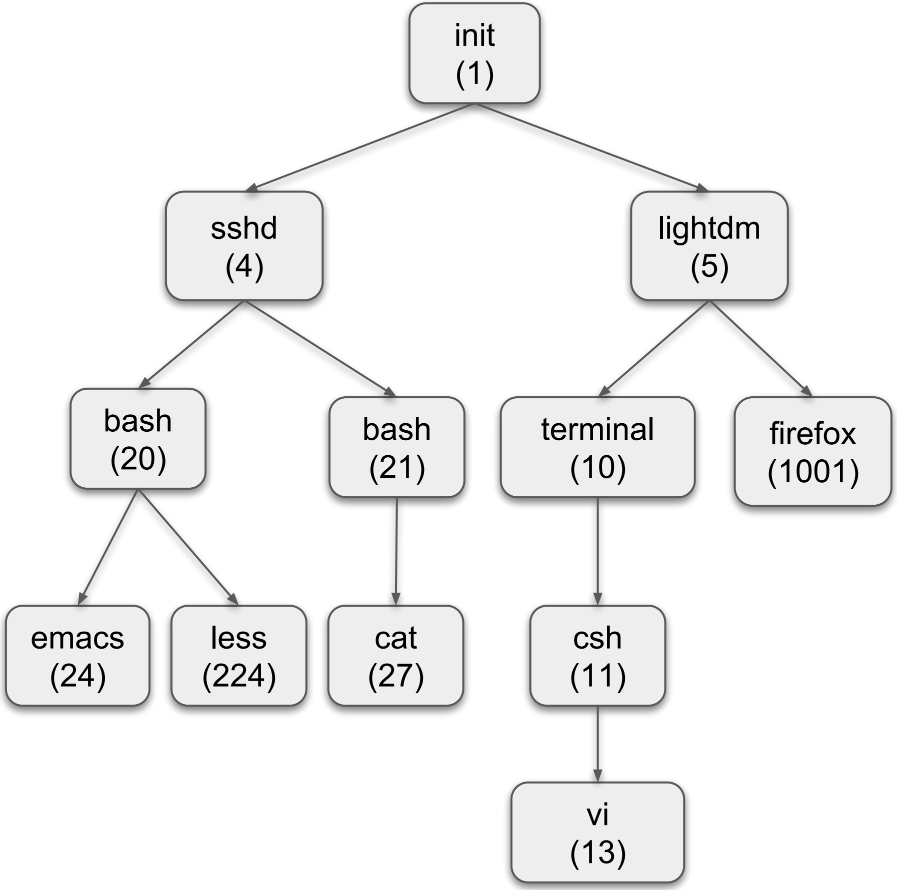
```
int main()
{
    pid_t  pid;

    pid = fork(); /* create & run child process - a duplicate of parent */
    /* both parent and child will execute the next line */
    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork failed");
        exit(-1);
    }
    else if (pid == 0) { /* child process */
        execlp("/bin/ls", "ls", "-l", NULL); /* replace process with 'ls -l' */
    }
    else { /* parent process will wait for the child to complete */
        printf("Waiting for child process %d\n", pid);
        wait(NULL);
        exit(0);
    }
}
```

A multiprocess program in C

```c
#include <stdlib.h>

int main()
{
    system("/bin/ls");
}
```

```
$ man system
...
The system() library function uses fork(2)
to create a child process that executes the
shell command specified in command using
execl(3) as follows:

    execl("/bin/sh", "sh", "-c", command,
          (char *) 0);

system() returns after the command has been
completed.
...
```

Process tree



- **parent process** - the creating process
- **child process (child)** - the newly created process
- **PID** - the unique process identifier for each process

- in Unix, parent and child processes continue to be associated, forming a process hierarchy

- in Windows, all processes are equal, the parent process can give the control of its children to any other process

init process

- `init` or `systemd` is the first process started after booting
  - older UNIX systems used init based systems
  - many popular Linux systems now switched to `systemd`
- `init` is the **ancestor** of all user processes (direct or indirect parent), i.e. root of process tree
- `init` always has PID = 1
- **orphaned** processes are adopted by `init`
- printing a process tree

```
$ pstree
$ ps axjf
```

- note: some special system 'processes' are created by kernel during bootstrap, and do not have to be descendants of init, such as `swapper` and `pagedaemon`

Review

- Which one of the following executes in kernel mode?
  - A user program
  - A library function call
  - A system call
  - A system call wrapper function

- In C, **`printf()`** is a system call.
  - True
  - False

Review

- When 4 programs are executing on a computer with a single CPU, how many program counters are there?
- When does a program become a process?

Review

- What is the name of the PCB data structure in Linux?
- Name some of fields in a PCB.
- On UNIX systems, what is the name of the process that is the ancestor of all user processes?

Summary

- system calls + related APIs
- processes
- processes implementation
  - PCB, address space
- fork, exec

Reference: 1.5, 1.6, 2.1.1, 2.1.2, 2.1.6 (Modern Operating Systems)

2.1 - 2.4, 3.1, 3.3 (Operating System Concepts)

Questions?