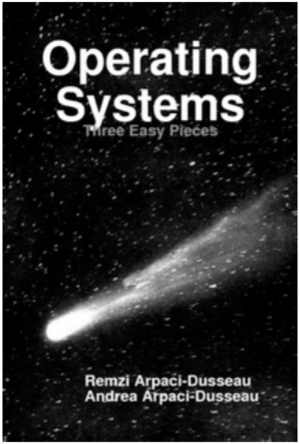
Operating Systems

Autumn 2024

OS abstraction: process

Textbook: "OSTEP"

Operating Systems: Three Easy Pieces (Available for free online)



Course topics

Virtualization

Virtualizing CPU
- Processes
- Threads
- Scheduling ...

Virtualizing Memory
- Address space
- Segmentation
- Paging ...

Concurrency

- Threads
- Locks
- Conditional variables
- Semaphores ...

Persistence

- File systems
- Journaling, logging
- I/O devices
- File integrity ...

Marking scheme

- Two in-sems: 25%
- End-sem: 50%
- Lab exercises: 15%
- Mini projects: 10%

Process != Program

- A process is a running instance of a program
    - Program = static file (image)
    - Process = executing program = program + execution state
- Several processes may run the same program code, but each is a distinct process with its own state
- A process executes sequentially, one instruction at a time
- Two processes are said to run concurrently when instructions of one process are interleaved with the instructions of the other process

Process states

- ready: waiting to be assigned to CPU
  - could run, but another process has the CPU

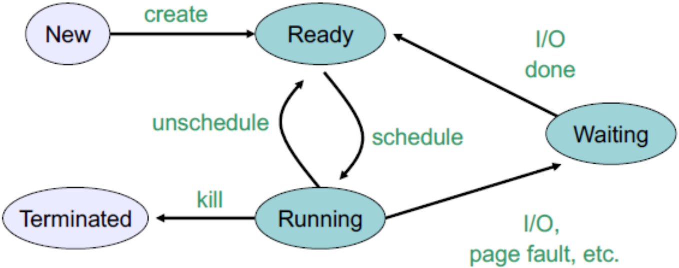
Process states

- ready: waiting to be assigned to CPU
    - could run, but another process has the CPU
- running: executing on the CPU
    - is the process that currently controls the CPU

Process states

- ready: waiting to be assigned to CPU
  - could run, but another process has the CPU
- running: executing on the CPU
  - is the process that currently controls the CPU
- waiting: waiting for an event, e.g. I/O
  - cannot make progress until event happens

Process state transitions



Question: What can cause schedule/unschedule transitions?

Example: state transitions

New
Ready
Running

```
main() {
   printf("Hello world");
}
```

Example: state transitions

New
Ready
Running

Waiting

```
main() {
   printf("Hello world");
}
```

Example: state transitions

```
main() {
   printf("Hello world");
}
```

New
Ready
Running

Waiting

Ready
Running
Terminated

Context Switch

- Switching the CPU from one process to another is called a context switch – relatively expensive operation
- Time sharing systems may do 100 to 1000 context switches a second

Creating a process

- One process can create other processes to do work
  - The creator is called the parent and the new process is the child
  - A parent can either wait for the child to complete, or continue in parallel

Creating a process

- One process can create other processes to do work
  - The creator is called the parent and the new process is the child
  - A parent can either wait for the child to complete, or continue in parallel
- In Unix, the fork() system call is used to create child processes

System calls

- OS procedures that perform privileged operations
  - Linux x86_64 has ∼323 system calls, numbered from 0–322
  - OS uses a sys_call_table to keep the syscall handlers (indexed by syscall number)
- Now process is able to perform ∼300 different kinds of restricted operations, and cannot access anything it wants to

Creating a process

- fork() copies variables and registers from the parent to the child
- fork(), when called, returns twice (to each process)
- Return value of fork()
  - In the parent process, fork() returns the process id of the child
  - In the child process, the return value is 0

Creating a process

- fork() copies variables and registers from the parent to the child
- fork(), when called, returns twice (to each process)
- Return value of fork()
  - In the parent process, fork() returns the process id of the child
  - In the child process, the return value is 0
- The parent can wait for the child to terminate by executing the wait system call or continue execution

pop quiz

```
main() {
  int x = 0;
  int cid = fork();
  if (cid == 0) {
     x = 9; printf("%d ", x);
  }
  else {
     x = 10; printf("%d ", x);
  }
}
```

This code will print out:

(A) 10 10

(B) 9 9

(C) 10 9

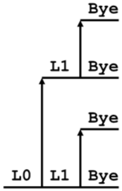
(D) 9 10

Caveats

- order of execution is non-deterministic
  - parent and child run concurrently
- Important: post fork, parent and child are identical but separate!
  - OS allocates and maintains separate data/state
  - control flow can diverge

Fun stuff: process graphs

```c
void fork2() {
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```



L1	L0	L0	L1	L0
L0	L1	L1	Bye	Bye
L1	Bye	Bye	Bye	Bye
Bye	Bye	Bye	L0	L1
Bye	L1	Bye	L1	L1
Bye	Bye	L1	Bye	Bye
Bye	Bye	Bye	Bye	Bye

Q

Okay, fork() creates a new process that is a duplicate of the process . . .

What if I want to run something different?

exec system call

- It does NOT create a new process
- It basically replaces the current process with a new program
- Does exec() ever return? If so, what does it mean?

  All functions (exec()-family) return -1 in the case of an error. Otherwise at successful execution there is no return back to the calling program. Thus, it is redundant to check the return value; you can directly continue with the error routine

pop quiz

Normally, I want to start a different program without replacing the current process (type "firefox" from a terminal).

How can we do this?

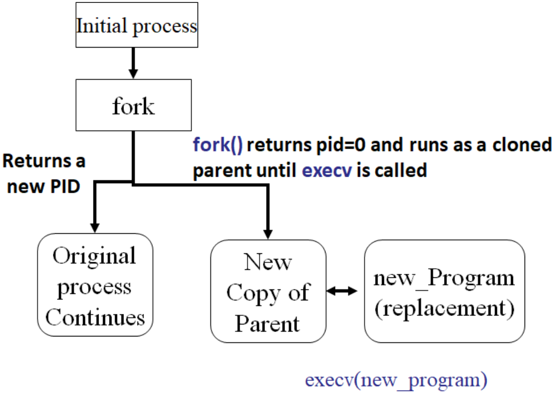
pop quiz

Normally, I want to start a different program without replacing the current process (type "firefox" from a terminal).

How can we do this?

Answer: call fork() then exec()

execv(new_program, argv[ ])



The scheduling problem

- Which process should the OS run?
  - if no runnable process (i.e. no process in the ready state), run the idle task
  - if a single process runable, run this one
  - if more than one runnable process, a scheduling decision must be taken
- Scheduler
  - code (logic) that decides which process to run as per some policy

Coming next

- What are the basic scheduling policies?
- When do they work well?