Processes

Sanzheng Qiao

Department of Computing and Software

December, 2012

What is a process?

The notion of process is an abstraction. It has been given many definitions. “Program in execution” is the most frequently referenced one.

Is a process the same as a program?

What is a process?

The notion of process is an abstraction. It has been given many definitions. “Program in execution” is the most frequently referenced one.

Is a process the same as a program?

No. It’s both more and less.

More: When a child process (e.g., *ls*) terminates, it signals its parent process (the shell). A process has the information about its parent/child processes.

Less: Program *cc* uses several processes.

Address space

Each process is associated with an *address space*:

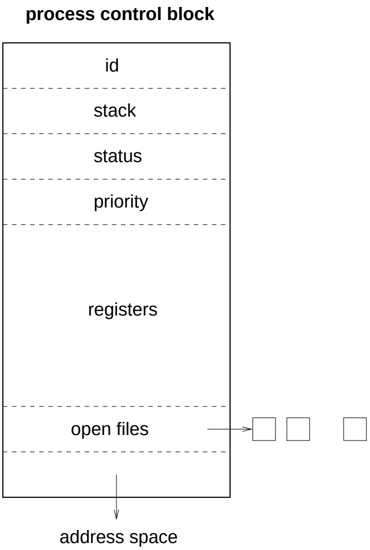
All the state needed to run a program (execution stack, system environment, etc.). It contains all the addresses that can be touched by the program.

Why address space: Protection. A process can only access its own address space.

A process is represented by its Process Control Block (PCB):

- Address space.
- Execution state (PC, saved registers).

PCB


process control block
id
stack
status
priority
registers
open files
address space

PCB

- Scheduling information (priority).
- Accounting information (CPU time).
- Open files.
- Other miscellaneous information.

OS maintains a process table (a collection of all PCBs) to keep track of all the processes.
In UNIX the process table is a fixed-size array.

Process states

New: Just created

Waiting: Waiting for an event to occur.

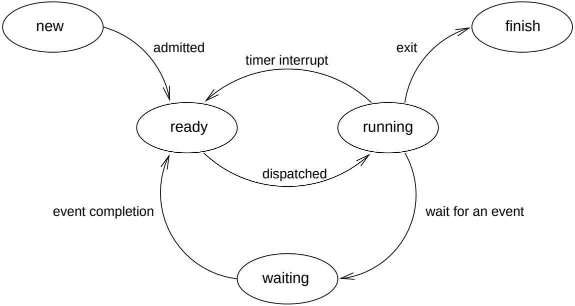
Ready: Has acquired all the resources but the CPU.

Running: Running on the CPU.

Finish: Exiting.

Processes switch from one state to another, OS controls this.

Process states



Dispatcher

With many processes on the the system, OS must take care of:

- scheduling: each process gets a fair share of the CPU time.
- protection: processes don’t modify each other.

Dispatcher

With many processes on the the system, OS must take care of:

- scheduling: each process gets a fair share of the CPU time.
- protection: processes don't modify each other.

Dispatcher:

1. Run process for a while
2. Pick a process from the ready queue
3. Save state (PC, registers, etc.)
4. Load state of next process
5. Run (load PC register)

Dispatcher

When a user process is switched out of the CPU, its state must be saved in its PCB. Everything could be damaged by the next process:

- Program counter.
- Processor status word.
- Registers (General purpose and floating-point).

Exceptions

The CPU can run only one at a time. When a user process is running, the dispatcher (part of OS) is not running.

How can OS regain control of the CPU?

- Exceptions: User process gives up the CPU to OS (caused by internal events, for example, go to sleep)
  - System call.
  - Error (eg. bus error, segmentation error, overflow, etc.).
  - Page fault.
  - Yield.

These are also called traps.

Interrupts

- Interrupts: The OS interrupts user process (caused by external events):
  - Completion of an input (eg. a character typed at keyboard)
  - Completion of an output (a character displayed at terminal)
  - Completion of a disk transfer
  - A packet is sent to the network.
  - Timer (alarm clock).

Process creation

Creating a process from scratch:

1. Load code and data into memory.
2. Set up a stack.
3. Initialize PCB.
4. Make process known to dispatcher.

Process creation

Forking a process:

1. Make sure the parent process is not running and has all state saved.
2. Make a copy of code, data, and stack.
3. Make a copy of PCB of the parent process into the child process.
4. Make the child process known to dispatcher.

Example

UNIX `fork()` and `exec()`.

The system call `fork()` is called by one process and returned in two processes.

Parent: returns child pid
Child: returns 0

```
pid = fork();
if (pid == 0)          /* child process */
    exec("executable");
/* parent process continues */
```

In the child process, `executable` overwrites the old program.

Process termination

Terminating when it finishes the last statement and calls `exit`.

- Deallocate memory (physical and virtual)
- Close open files
- Notify its parent process

Process termination

Terminated by another process, usually the parent, using system call `abort` or `kill`.

- The child has exceeded some resource quota
- The child's task is no longer needed
- The parent is exiting