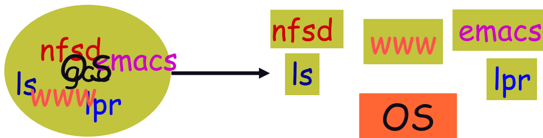
Processes

Announcements

Why Processes? Simplicity + Speed

- Hundreds of things going on in the system



- How to make things simple?
  - Separate each in an isolated process
  - Decomposition
- How to speed-up?
  - Overlap I/O bursts of one process with CPU bursts of another

What is a process?

- The unit of execution
- The unit of scheduling
- Thread of execution + address space
- Is a program in execution
  - Sequential, instruction-at-a-time execution of a program.

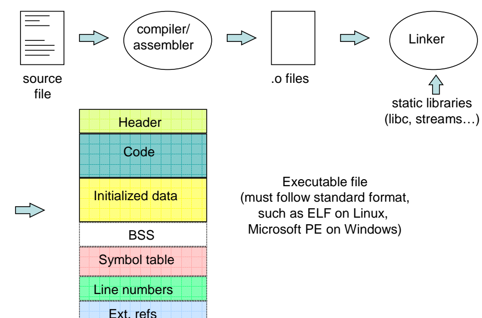
The same as “job” or “task” or “sequential process”

What is a program?

A program consists of:

- **Code:** machine instructions
- **Data:** variables stored and manipulated in memory
  - initialized variables (globals)
  - dynamically allocated variables (malloc, new)
  - stack variables (C automatic variables, function arguments)
- **DLLs:** libraries that were not compiled or linked with the program
  - containing code & data, possibly shared with other programs
- **mapped files:** memory segments containing variables (mmap())
  - used frequently in database programs

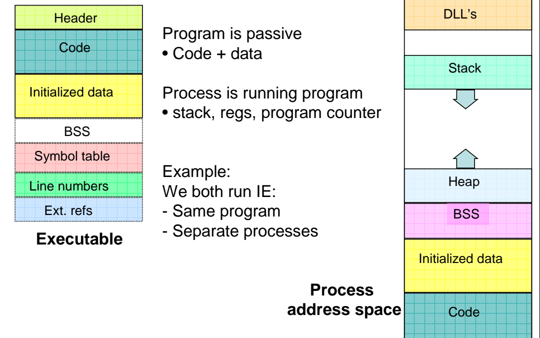
Preparing a Program



Running a program

- OS creates a "process" and allocates memory for it
- The loader:
  - reads and interprets the executable file
  - sets process's memory to contain code & data from executable
  - pushes "argc", "argv", "envp" on the stack
  - sets the CPU registers properly & calls "__start()" [Part of CRT0]
- Program start running at __start(), which calls main()
  - we say "process" is running, and no longer think of "program"
- When main() returns, CRT0 calls "exit()"
  - destroys the process and returns all resources

Process != Program



Program is passive
- Code + data

Process is running program
- stack, regs, program counter

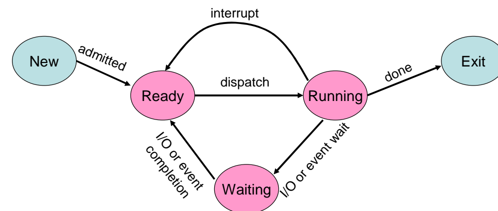
Example:
We both run IE:
- Same program
- Separate processes

Process address space

Process States

- Many processes in system, only one on CPU
- "Execution State" of a process:
  - Indicates what it is doing
  - Basically 3 states:
    - Ready: waiting to be assigned to the CPU
    - Running: executing instructions on the CPU
    - Waiting: waiting for an event, e.g. I/O completion
- Process moves across different states

Process State Transitions



Processes hop across states as a result of:
- Actions they perform, e.g. system calls
- Actions performed by OS, e.g. rescheduling
- External actions, e.g. I/O

Process Data Structures

- OS represents a process using a *PCB*
  - Process Control Block
  - Has all the details of a process

Process Id	Security Credentials
Process State	Username of owner
General Purpose Registers	Queue Pointers
Stack Pointer	Signal Masks
Program Counter	Memory Management
Accounting Info	…

Context Switch

- For a running process
  - All registers are loaded in CPU and modified
    - E.g. Program Counter, Stack Pointer, General Purpose Registers
- When process relinquishes the CPU, the OS
  - Saves register values to the PCB of that process
- To execute another process, the OS
  - Loads register values from PCB of that process

⇒ **Context Switch**
  - Process of switching CPU from one process to another
  - Very machine dependent for types of registers

Details of Context Switching

- Very tricky to implement
  - OS must save state without changing state
  - Should run without touching any registers
    - CISC: single instruction saves all state
    - RISC: reserve registers for kernel
      - Or way to save a register and then continue
- Overheads: CPU is idle during a context switch
  - Explicit:
    - direct cost of loading/storing registers to/from main memory
  - Implicit:
    - Opportunity cost of flushing useful caches (cache, TLB, etc.)
    - Wait for pipeline to drain in pipelined processors

How to create a process?

- Double click on a icon?
- After boot OS starts the first process
  - E.g. sched for Solaris, ntoskrnel.exe for XP
- The first process creates other processes:
  - the creator is called the parent process
  - the created is called the child process
  - the parent/child relationships is expressed by a process tree
- For example, in UNIX the second process is called *init*
  - it creates all the gettys (login processes) and daemons
  - it should never die
  - it controls the system configuration (#processes, priorities…)
- Explorer.exe in Windows for graphical interface

Processes Under UNIX

- Fork() system call is only way to create a new process
- int fork() does many things at once:
  - creates a new address space (called the child)
  - copies the parent's address space into the child's
  - starts a new thread of control in the child's address space
  - parent and child are equivalent -- almost
    - in parent, fork() returns a non-zero integer
    - in child, fork() returns a zero.
    - difference allows parent and child to distinguish
- int fork() returns TWICE!

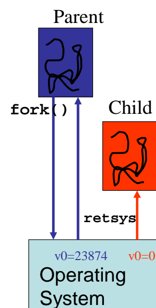
Example

```
main(int argc, char **argv)
{
   char *myName = argv[1];
   int cpid = fork();
   if (cpid == 0) {
      printf("The child of %s is %d\n", myName, getpid());
      exit(0);
   } else {
      printf("My child is %d\n", cpid);
      exit(0);
   }
}
```

What does this program print?

Bizarre But Real

```
lace:tmp<15> cc a.c
lace:tmp<16> ./a.out foobar
The child of foobar is 23874
My child is 23874
```



Fork is half the story

- Fork() gets us a new address space,
  - but parent and child share EVERYTHING
    - memory, operating system state
- int exec(char *programName) completes the picture
  - throws away the contents of the calling address space
  - replaces it with the program named by programName
  - starts executing at header.startPC
  - Does not return
- Pros: Clean, simple
- Con: duplicate operations

Starting a new program

```
main(int argc, char **argv)
{
   char *myName = argv[1];
   char *progName = argv[2];

   int cpid = fork();
   if (cpid == 0) {
      printf("The child of %s is %d\n", myName, getpid());
      execlp("/bin/ls",   // executable name
             "ls", NULL); // null terminated argv
      printf("OH NO. THEY LIED TO ME!!!\n");
 } else {
      printf("My child is %d\n", cpid);
      exit(0);
   }
}
```

Process Termination

- Process executes last statement and OS decides(**exit**)
  - Output data from child to parent (via **wait**)
  - Process' resources are deallocated by operating system
- Parent may terminate execution of child process (**abort**)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent is exiting
    - Some OSes don't allow child to continue if parent terminates
      - All children terminated - *cascading termination*

ProcExp Demo

- Windows process hierarchy
- explorer.exe and the system idle process
- Windows base priority mechanism
  - 0, 4, 8, 13, 24
  - What is procexp's priority?
- Creating a new process
- Terminating a process