Operating Systems Design (CS 423)

Elsa L Gunter
2112 SC, UIUC
http://www.cs.illinois.edu/class/cs423/

Based on slides by Roy Campbell, Sam King, and Andrew S Tanenbaum

Unix System Calls

- Process management:
  - fork, exec, wait
- File handling
  - open, read, write, close
- File namespace
  - readdir, link, unlink, rename

Implement a Shell

- Basic behavior
  - Wait for input; parse it to get command
  - Create child process; wait for it to return
  - Repeat
- Tools to use:
  - fork(): makes two processes with same code; returns new pid (of child) to parent, 0 to child
  - exec(argv): replace current process with argv (approximation)
  - waitpid(pid,statloc,op): wait for process with pid to return

System call: execve

- int execve(const char *filename, char *const argv [], char *const envp[]);
- int execve(const char *filename, char *const * argv , char * const * envp);
- filename: string giving full path to executable file
- argv[]: a null-terminated array of character pointers (strings giving command line args)
  - Should have argv[0] and filename give same file
- envp[]: a null-terminated array of character pointers (strings giving environment vector)

Implement a Shell

```
char argv[] ; int statloc ;
while(1) {
  char* user_input = display_prompt();
  parse_input(user_input,&argv);




}
```


Implement a Shell (approximation)

```
char argv[]; int statloc;
while(1) {
  char* user_input = display_prompt();
  parse_input(user_input,&argv);
  pid = fork();
  if (pid == 0) //child process
  {execve(argv[0], &argv, 0)}
  else {waitpid(pid, &statloc, 0)}
          //parent
}
```

Process Illusion

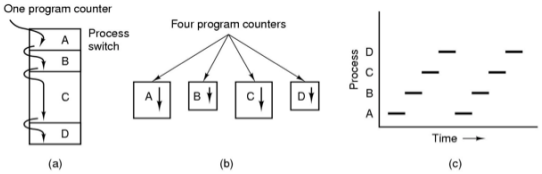
- Illusion: infinite number of processes, exclusive access to CPU, exclusive access to infinite memory
- Reality: fixed process table sizes, shared CPUs, finite shared memory
- Can break illusion: create a process that creates a process that creates a process …
- Will lock up your system, may need to reboot

Fork bomb

- How can we fix this?
- What price do we pay for the fix?
- Is it worth it to fix it?
- Two perspectives: Math and engineering

Why Processes?

- Heart of modern (multiprocess) OS



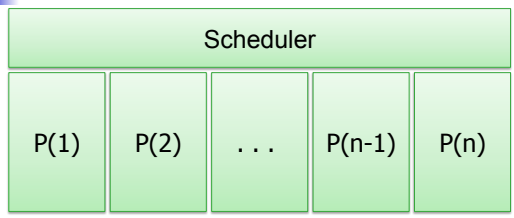
a) Multiprogramming of four programs.
b) Conceptual model of four independent, sequential processes.
c) Only one program is active at once.

Why Processes?

- Processes decompose mix of activities running on a processor into several parallel tasks

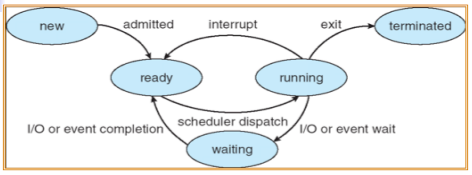
- Each job can work independently of the others
- Remember, for any area of OS, ask:
  - What interface does the hardware provide?
  - What interface does the OS provide?

Approx Implementation of Processes



- Scheduler interleaves processes, preserving their state
- Abstraction: sequential processes on dedicated CPU

Process States



- As a process executes, it changes *state*
  - new: Process being created
  - ready: Process waiting to run
  - running: Instructions are executed
  - waiting: Process waiting for some event to occur
  - terminated: Process finished execution

What is a process?

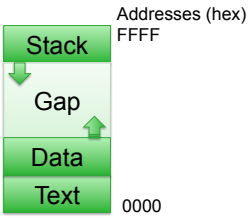
- Process definition (old view): instance of execution of a program
  - A running piece of code with all resources it can read / write
  - Note! Program =/= process
- Process definition (modern view): Full collection of threads sharing same address space
- Play analogy: Process is a performance of a play on a stage in a stage house

Process Components

- Thread
  - Sequence of executing instructions from a program (i.e., the running computation)
  - Active
  - Play analogy: actors on stage
- Address space
- All data used by process as it runs
- Passive (acted upon by the thread)
- Play analogy: stage, possibly stage props

UNIX Process Memory Layout

- Process has three segments
  - Text – lowest addresses, fixed size
  - Data – immediately after text, grows only by explicitly system call
  - Stack – top of region, grows downward automatically
  - Gap in middle for dynamic allocation



Stack

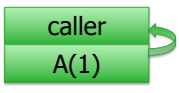
```
void A(int tmp){
if (tmp == 2) {return}
else    {B()};
}
void B(){
   C();
}
void C(){
   A(2);
}
Start: A(1);
```

- Stack

Stack

```
void A(int tmp){
if (tmp == 2) {return}
else    {B()};
}
void B(){
   C();
}
void C(){
   A(2);
}
Start: A(1);
```

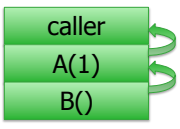
- Stack



Stack

```
void A(int tmp){
if (tmp == 2) {return}
else    {B()};
}
void B(){
   C();
}
void C(){
   A(2);
}
Start: A(1);
```

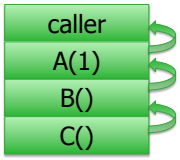
- Stack



Stack

```
void A(int tmp){
if (tmp == 2) {return}
else    {B()};
}
void B(){
   C();
}
void C(){
   A(2);
}
Start: A(1);
```

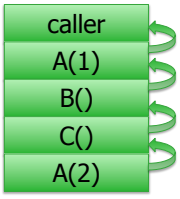
- Stack



Stack

```
void A(int tmp){
if (tmp == 2) {return}
else    {B()};
}
void B(){
   C();
}
void C(){
   A(2);
}
Start: A(1);
```

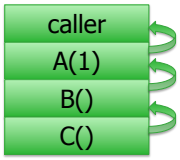
- Stack



Stack

```
void A(int tmp){
if (tmp == 2) {return}
else    {B()};
}
void B(){
   C();
}
void C(){
   A(2);
}
Start: A(1);
```

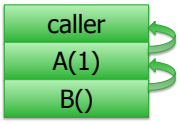
- Stack



Stack

```
void A(int tmp){
if (tmp == 2) {return}
else    {B()};
}
void B(){
   C();
}
void C(){
   A(2);
}
Start: A(1);
```

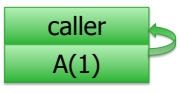
- Stack



Stack

```
void A(int tmp){
if (tmp == 2) {return}
else    {B()};
}
void B(){
   C();
}
void C(){
   A(2);
}
Start: A(1);
```

- Stack



Stack

```
void A(int tmp){
if (tmp == 2) {return}
else    {B()};
}
void B(){
   C();
}
void C(){
   A(2);
}
Start: A(1);
```

- Stack

Multiple threads

- Can have several threads in a single address space
  - Play analogy: several actors on a single set. Sometimes interact (e.g., dance together), sometimes do independent tasks
- Private state for a thread vs. global state shared between threads