Process Description and Control in Unix

B. Ramamurthy

Fall 2006

Creation of a process

- A unique pid is assigned to the new process.
- Space is allocated for all the elements of the process image.
- The process control block is initialized. Inherit info from parent.
- The appropriate linkages are set: for scheduling, state queues..
- Create and initialize other data structures (file tables, IO table etc.).

Process Interruption

- Two kinds of process interruptions: **interrupt** and **trap**.
- **Interrupt:** Caused by some event external to and asynchronous to the currently running process, such as completion of IO.
- **Trap** : Error or exception condition generated within the currently running process. Ex: illegal access to a file, arithmetic exception.
- (supervisor call) : explicit interruption.

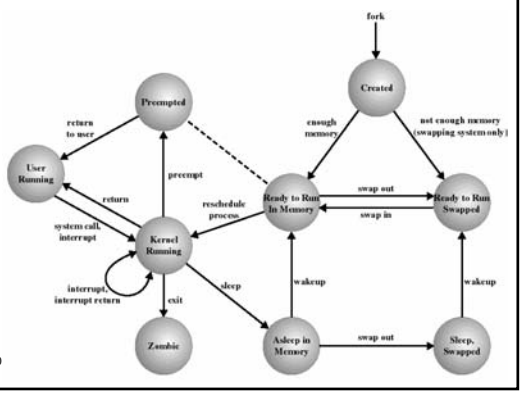
Unix system V

- All user processes in the system have as root ancestor a process called **init.** When a new interactive user logs onto the system, **init** creates a user process, subsequently this user process can create child processes and so on. init is created at the boot-time.
- Process states : User running , kernel running, Ready in memory, sleeping in memory (blocked), Ready swapped (ready-suspended), sleeping swapped (blocked-suspended), created (new), zombie , preempted (used in real-time scheduling).

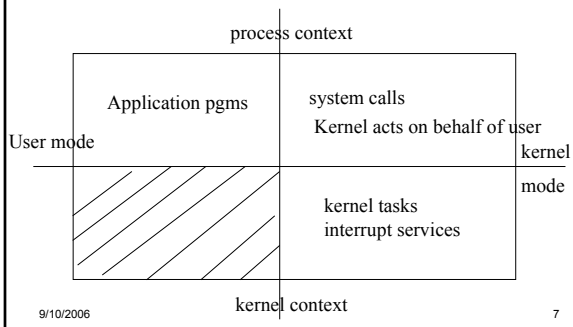
UNIX SVR4 Process States

- Similar to our 7 state model
- 2 running states: User and Kernel
  - transitions to other states (blocked, ready) must come from kernel running
- Sleeping states (in memory, or swapped) correspond to our blocking states
- A preempted state is distinguished from the ready state (but they form 1 queue)
- Preemption can occur only when a process is about to move from kernel to user mode

UNIX Process State Diagram



Process and kernel context



Unix system V (contd.)

- What does unix process image contain?
- What does process table entry contain? **proc**
- What is unix U (user) area? **u area**
- Function of each of these components.

U area

- Process control block
- Pointer to proc structure (process table entry)
- Signal handlers related information
- Memory management information
- Open file descriptor
- Vnodes(?) of the current directory
- CPU usage stats
- Per process kernel stack

Process Context

- User address space,
- Control information : u area (accessed only by the running process) and process table entry (or proc area, accessed by the kernel)
- Credentials : UID, GID etc.
- Environment variables : inherited from the parent

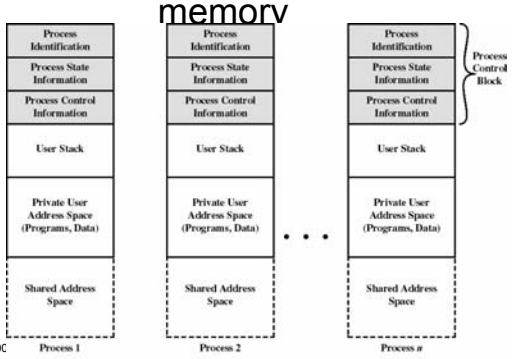
UNIX Process Image

- User-level context
  - Process Text (ie: code: read-only)
  - Process Data
  - User Stack (calls/returns in user mode)
  - Shared memory (for IPC)
    - only one physical copy exists but, with virtual memory, it appears as it is in the process's address space
- Register context

UNIX Process Image

- System-level context
  - Process table entry
    - the actual entry concerning this process in the Process Table maintained by OS
      - Process state, UID, PID, priority, event awaiting, signals sent, pointers to memory holding text, data...
  - U (user) area
    - additional process info needed by the kernel when executing in the context of this process
      - effective UID, timers, limit fields, files in use ...
  - Kernel stack (calls/returns in kernel mode)
  - Per Process Region Table (used by memory manager)

Process images in virtual memory



Process control

- Process creation in unix is by means of the system call fork().
- OS in response to a fork() call:
  - Allocate slot in the process table for new process.
  - Assigns unique pid.
  - Makes a copy of the process image, except for the shared memory.
  - Move child process to Ready queue.
  - **it returns pid of the child to the parent, and a zero value to the child**.

Process control (contd.)

- All the above are done in the kernel mode in the process context. When the kernel completes these it does one of the following as a part of the dispatcher:
  - Stay in the parent process. Control returns to the user mode at the point of the fork call of the parent.
  - Transfer control to the child process. The child process begins executing at the same point in the code as the parent, at the return from the fork call.
  - Transfer control another process leaving both parent and child in the Ready state.

UNIX Process Creation

- Every process, except process 0, is created by the fork() system call
  - fork() allocates entry in process table and assigns a unique PID to the child process
  - child gets a copy of process image of parent: both child and parent are executing the same code following fork()
  - but fork() returns the PID of the child to the parent process and returns 0 to the child process

Process creation - Example

```
main () {
  int pid;
  cout << " just one process so far"<<endl;
  pid = fork();
  if (pid == 0)
    cout <<"I am the child "<< endl;
  else if (pid > 0)
     cout <<"I am the parent"<< endl;
   else
     cout << "fork failed"<< endl;}
```


fork and exec

- Child process may choose to execute some other program than the parent by using exec call.
- Exec overlays a new program on the existing process.
- Child will not return to the old program unless exec fails. This is an important point to remember.
- Why does fork need to clone?
- Why do we need to separate fork and exec?
- Why can't we have a single call that fork a new program?

Example

```
if (( result = fork()) == 0 ) {
   // child code
  if (execv (“new program”,..) < 0)
     perror (“execv failed “);
     exit(1);
}
else if (result < 0 ) perror (“fork”); …}
/* parent code */
```


Version of exec

- Many versions of exec are offered by C library: exece, execve, execvp,execl, execle, execlp
- We will look at these and methods to synchronize among various processes (wait, signal, exit etc.).