Operating Systems

Processor Virtualization via Process Abstraction (Lecture-2)

Monsoon 2014, IIIT-H, Suresh Purini

Virtualizing a Processor

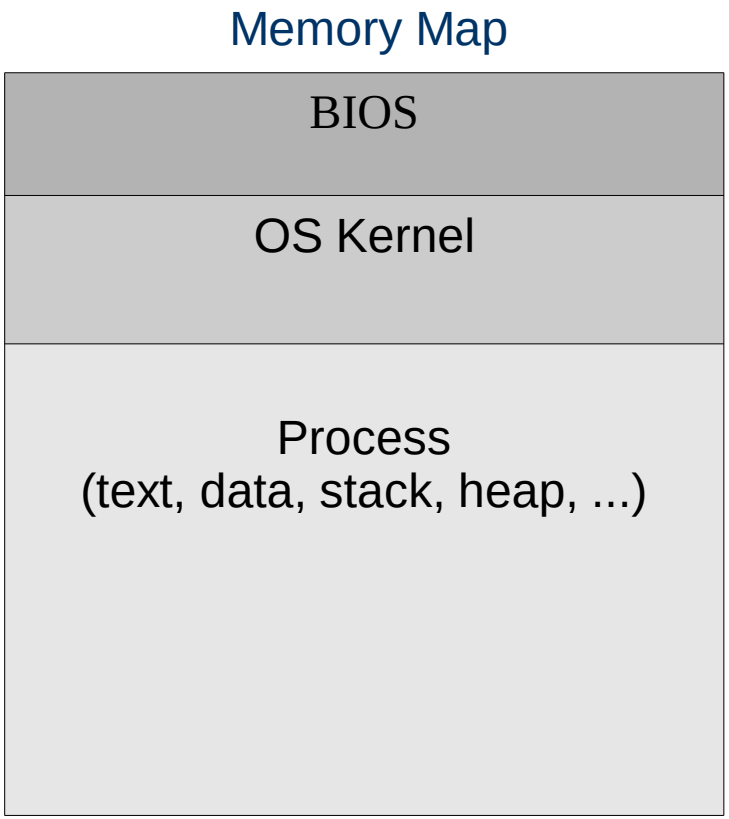
- Goal
  - We would like to run multiple programs on a single CPU (may be more).
  - Each program should get an illusion that it has the entire CPU for itself.
    - In other words, provide each program a separate virtual CPU.
- Possible Tradeoff: Virtual CPU may run a bit slower than the underlying Physical CPU.
- Idea: A program relinquishes the CPU in either of the following two scenarios
  - Has to wait on an I/O request
  - Already used the allocated time slice (time multiplexing)

Process Abstraction

- A process is a dynamic object which can be created and destroyed using an OS API (aka system calls).
- The state of the process and its dynamics (or evolution) are a function of the static program from which it is created.
- Simply Put: A process is a program in execution.
  - A program is a static entity whereas

    a process is a dynamic entity.
- Process State: text/code, data, bss, stack, heap,

  CPU registers, program counter, …, anything else!
- As a process executes its state keep evolving.

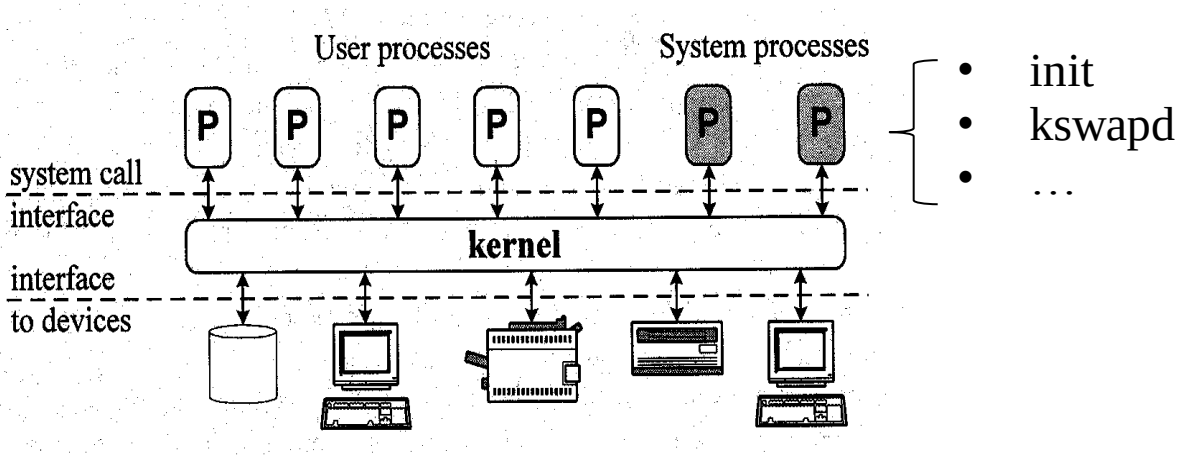


Per Process Kernel Data Structures

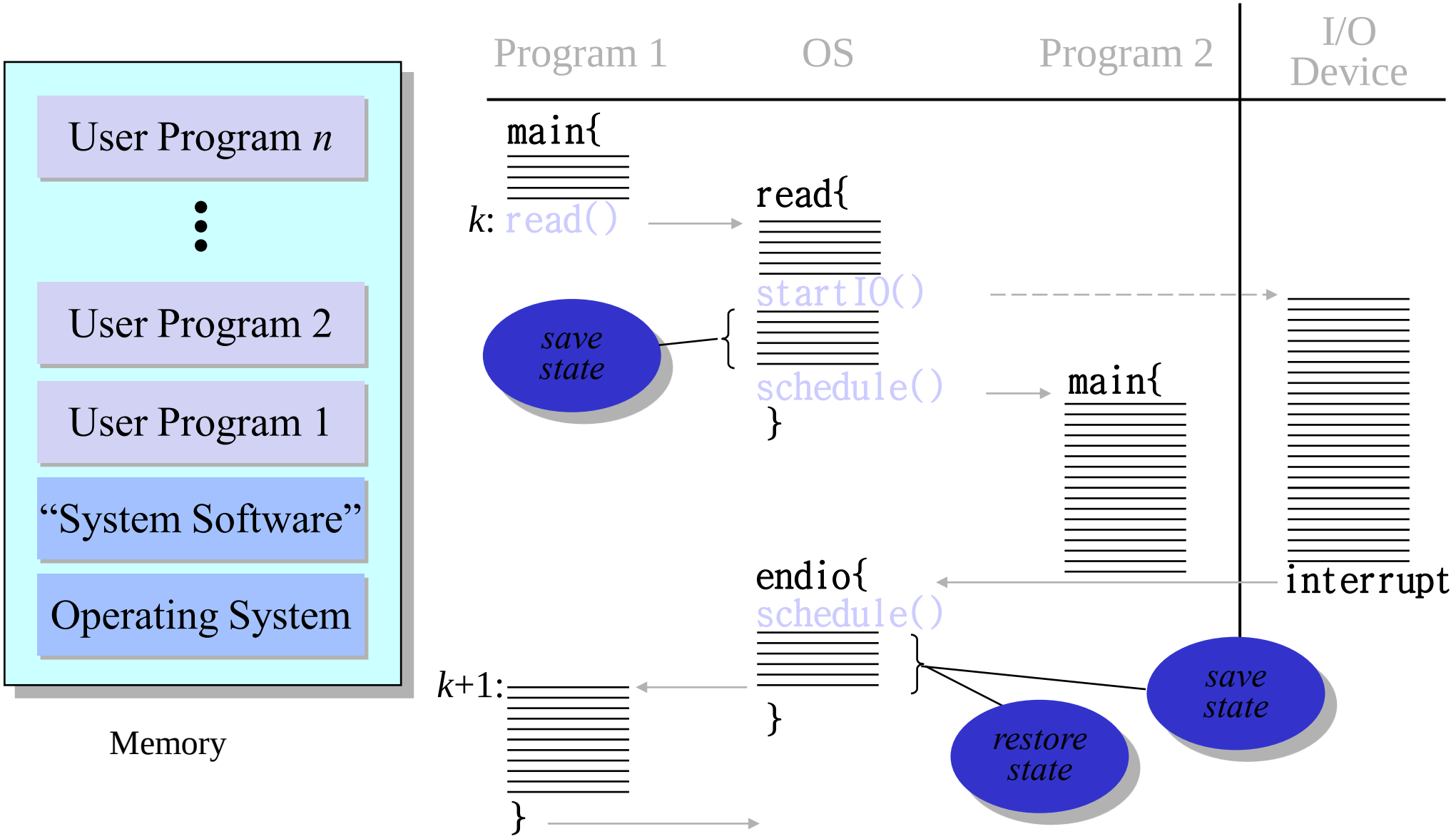
- u-area (accessible only when the process is currently running)
  - Process control block – Stores the hardware context when the process is not running
  - Open file descriptor tables
  - CPU and other resource usage statistics
  - Per kernel stack
  - ……
- proc structure (accessible at any time)
  - Process id
  - Location of the kernel address map for the u-area of this process
  - Current process state (like READY, STOPPED, BLOCKING, …)
  - ……..

Process Abstraction

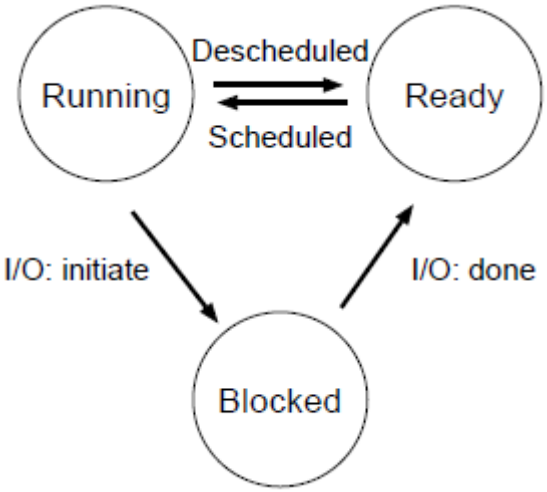
- Every process should get
  - a virtual CPU for itself
  - a complete virtual memory address space
- Different processes may run different instances of the same program
  - E.g., Two Firefox processes are fired from the same program.
- Every process gets a unique identifier (pid)

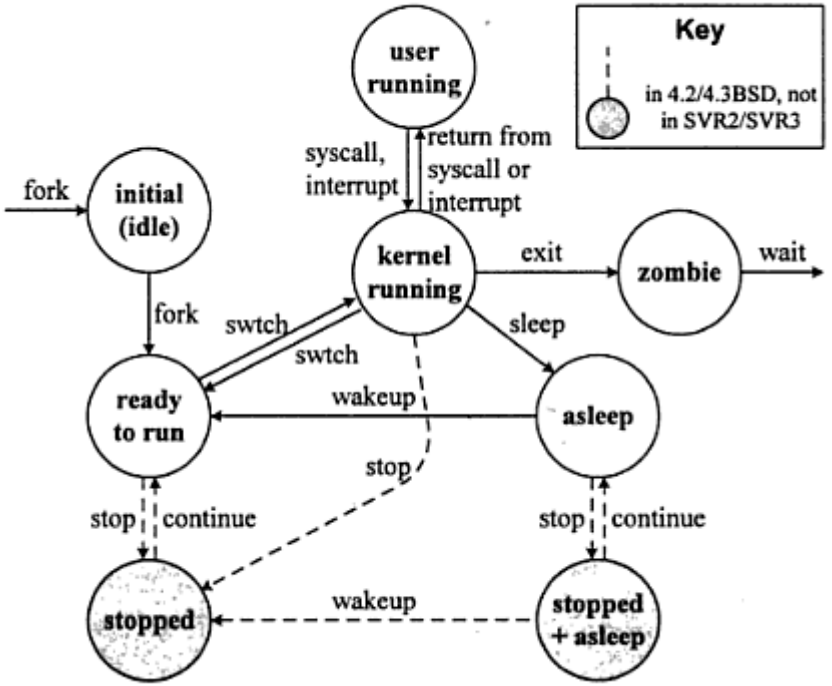


Context Switching



Process State Transitions





Scheduling Policy: Among the available ready-to-run processes which one should be allocated CPU in the next time slice.

Unresolved Issues

1. Kernel isolation
2. Process isolation
3. Address space isolation

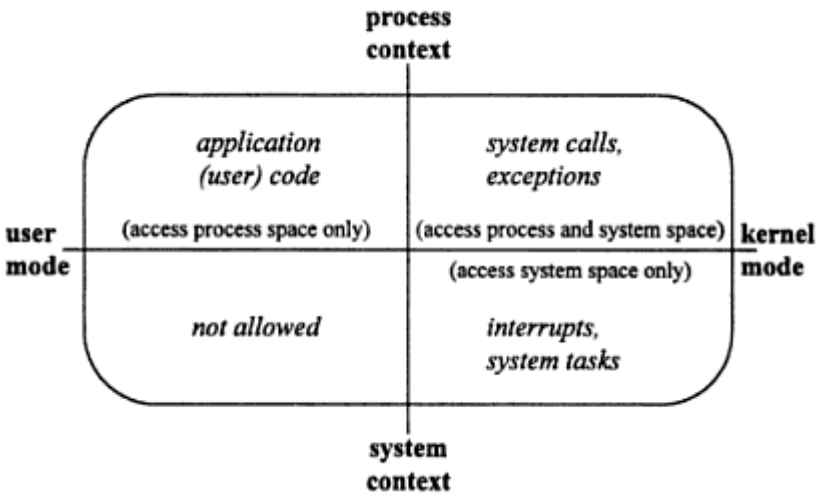
Tying it All Together: The Unix Shell

```
while(! EOF) {
read input
handle regular expressions
int pid = fork();           // create a child
if(pid == 0) {              // child continues here
    exec("program", argc, argv0, argv1, …);
}
else {                  // parent continues here
…
}
```

- Translates <CTRL-C> to the kill() system call with SIGKILL
- Translates <CTRL-Z> to the kill() system call with SIGSTOP
- Allows input-output redirections, pipes, and a lot of other stuff that we will see later

Mode, Space and Context

- User Mode vs Kernel Mode
  - Realized using processor modes
- User Space vs System Space
- Process Context vs System Context


process
context
application
(user) code
system calls,
exceptions
user
mode
(access process space only)
(access process and system space)
kernel
mode
(access system space only)
not allowed
interrupts,
system tasks
system
context

Preemptive versus Nonpreemptive