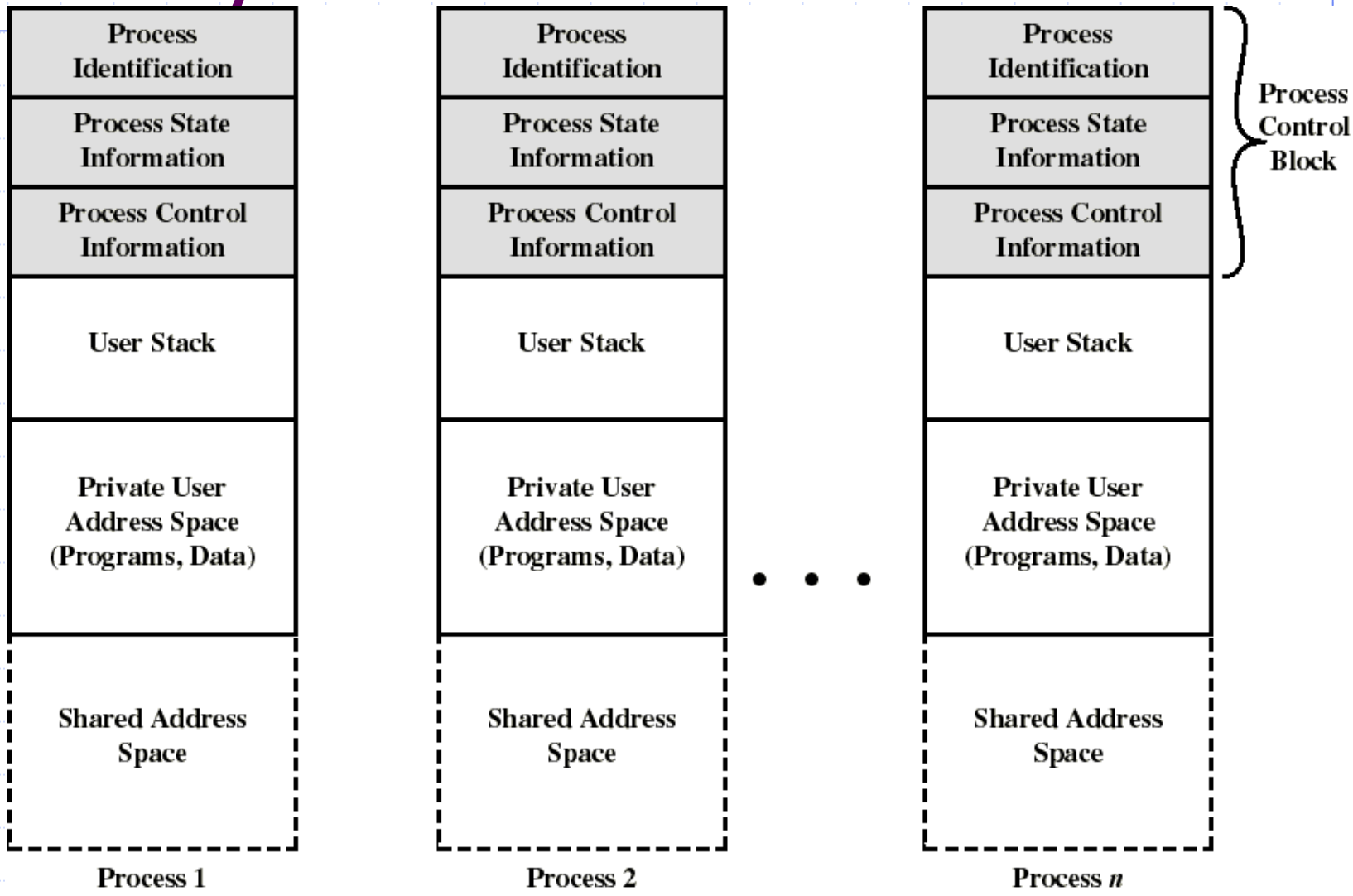
Unix Process Control

B.Ramamurthy

Process images in virtual memory



Creation of a process

- Assign a unique pid to the new process.
- Allocate space for all the elements of the process image. How much?
- 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.

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.

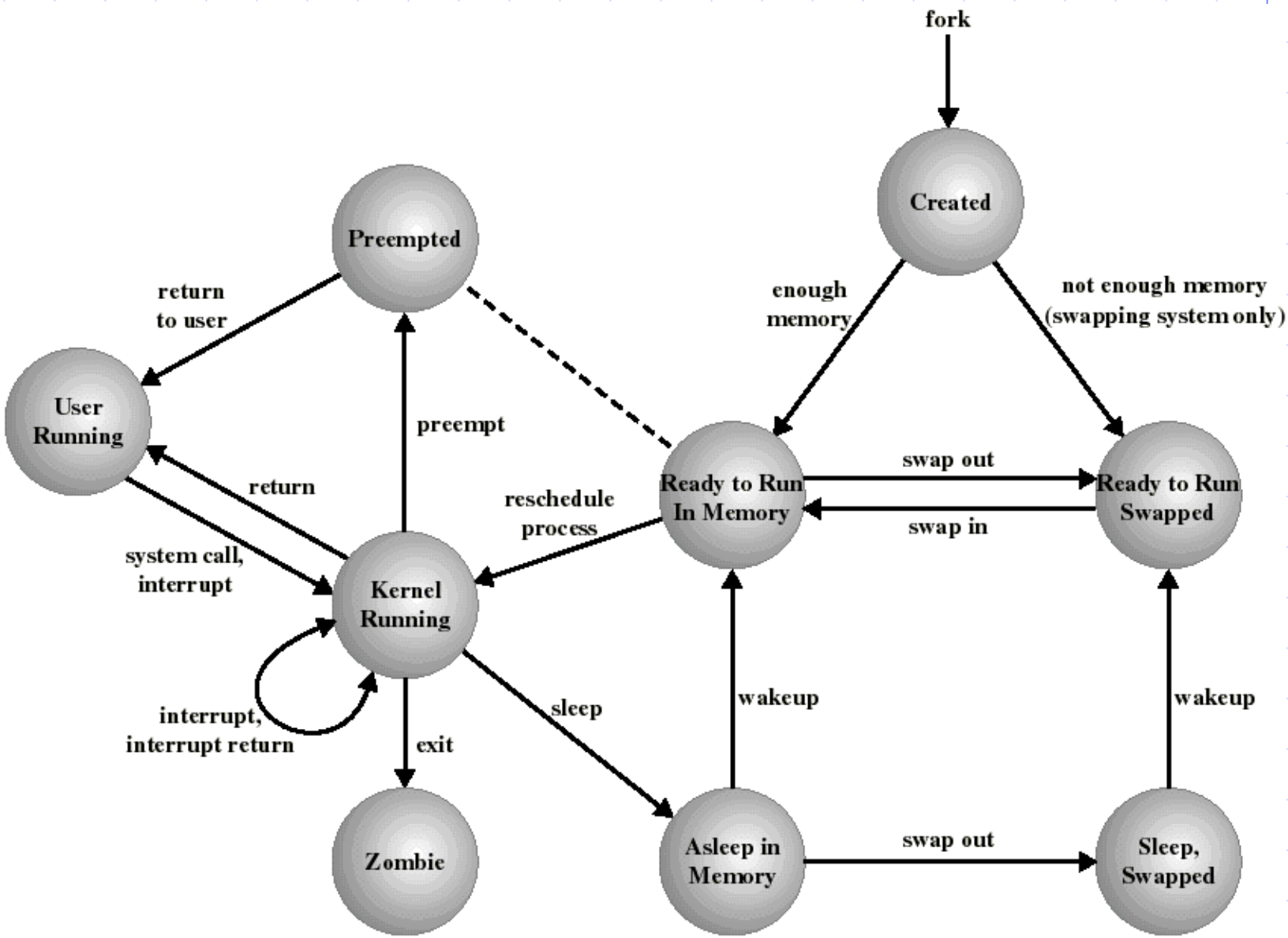
Unix system V

- 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 Context Switching

- Clock interrupt: The OS determines if the time slice of the currently running process is over, then switches it to Ready state, and dispatches another from Ready queue. "Process switch"
- Memory fault: (Page fault) A page fault occurs when the requested program page is not in the main memory. OS (page fault handler) brings in the page requested, resumes faulted process.
- IO Interrupt : OS determines what IO action occurred and takes appropriate action.

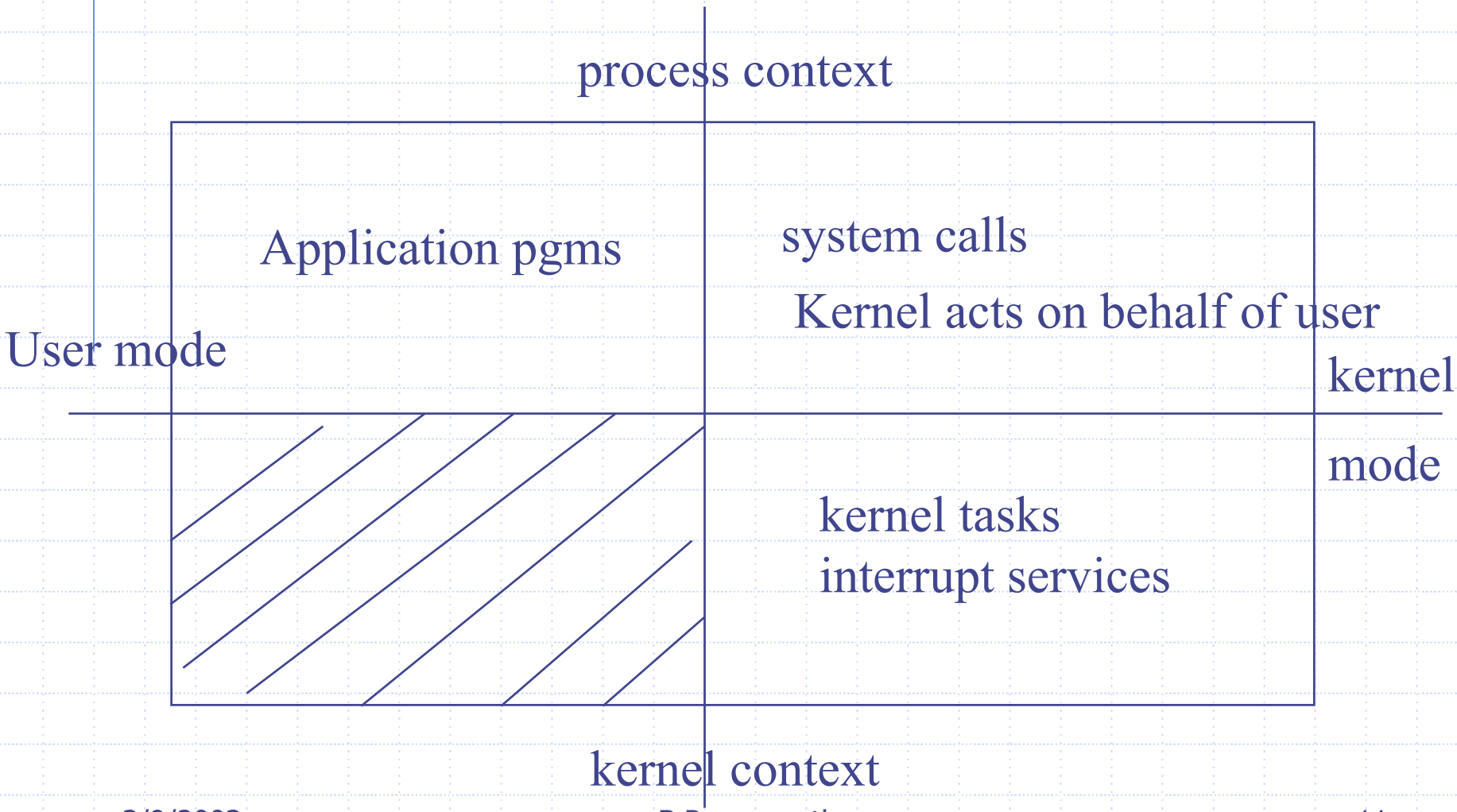
Process and Context Switching (contd.)

- How many context switch occurs per process switch?
- Typically 1 Process switch : 100 context switches
- Process switch of more expensive than context switch.
- Read more on this.
- This factor is very important for many system design projects.

Process and Context Switching (contd.)

- Process switch: A transition between two memory-resident processes in a multiprogramming environment.
- Context switch: Changing context from a executing program to an Interrupt Service Routine (ISR). Part of the context that will be modified by the ISR needs to be saved. This required context is saved and restored by hardware as specified by the ISR.

Process and kernel context



U area

- Process control block
- Pointer to proc structure
- 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 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

UNIX System Processes

- Process 0 is created at boot time and becomes the “swapper” after forking process 1 (the INIT process)
- When a user logs in: process 1 creates a process for that user

Process creation - Example

```
main () {
  int pid;
  cout << " just one process so far"<<endl;
  pid = fork();
  if (pid == 0)
    cout <<"im the child "<< endl;
  else if (pid > 0)
     cout <<"im 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 do we need to separate fork and exec? Why can't we have a single call that fork a new program?
- See the enclosed sheets from Havilland and Salama's text.

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 */
```