Process States and Transitions

Process States

- The Process is **executing** in **user mode**
- The Process is **executing** in **kernel mode**
- The Process is **not executing** but is **ready to run** as soon as the kernel schedules it (in the ready queue).
- The process is **sleeping** and **resides in the main memory**.
- The process is **ready to run**, but the **swapper must swap the process into main memory** before the kernel can schedule it to execute (is ready to run but swapped out of the main memory).
- The process is **sleeping**, and the **swapper has swapped the process into secondary memory** to make room for the other processes in main memory (sleeping and swapped out).
- The process is **returning from the kernel to user mode**, but the **kernel preempts it** and does a context switch to schedule another process.
- The process is **newly created** and is in a transition state (**process exists but not ready to run** and nor it is sleeping).
- The process **executed** `exit()` and is in a **zombie state** (the process no longer exists but it leaves a record containing an exit code and some time statistics for its parent process to collect), which is a final state of a process.

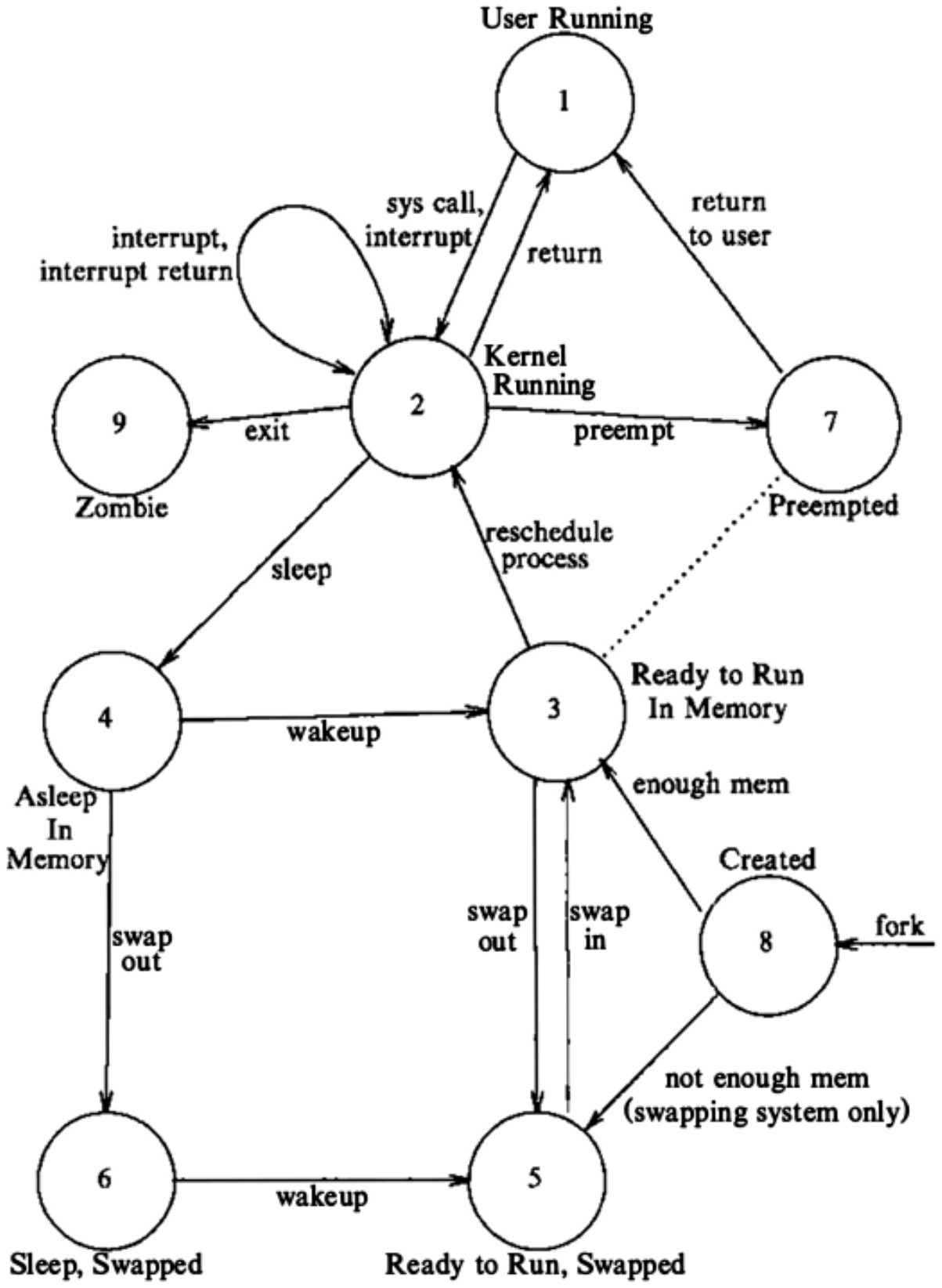


Figure 6.1. Process State Transition Diagram

Contents of process table

- Several UIDs that determine various priviledges.
- Process IDs or PIDs
- Event descriptors when the process is in *sleep* state.
- Scheduling parameters that allow kernel to determine which process move to the states *user running* and *kernel running*.
- Signal field
- Various timers.

Contents of user area (u area)

- Pointer to process table
- Real and effective UIDs.
- Timer fields.
- An array indicates how processes wishes to react to kernels.
- Control terminal field (login terminal associated with it).
- An error field
- Return value field
- I/O parameters.
- current directory and current root
- Limit fields to restrict the size of a process.
- Permission mode fields

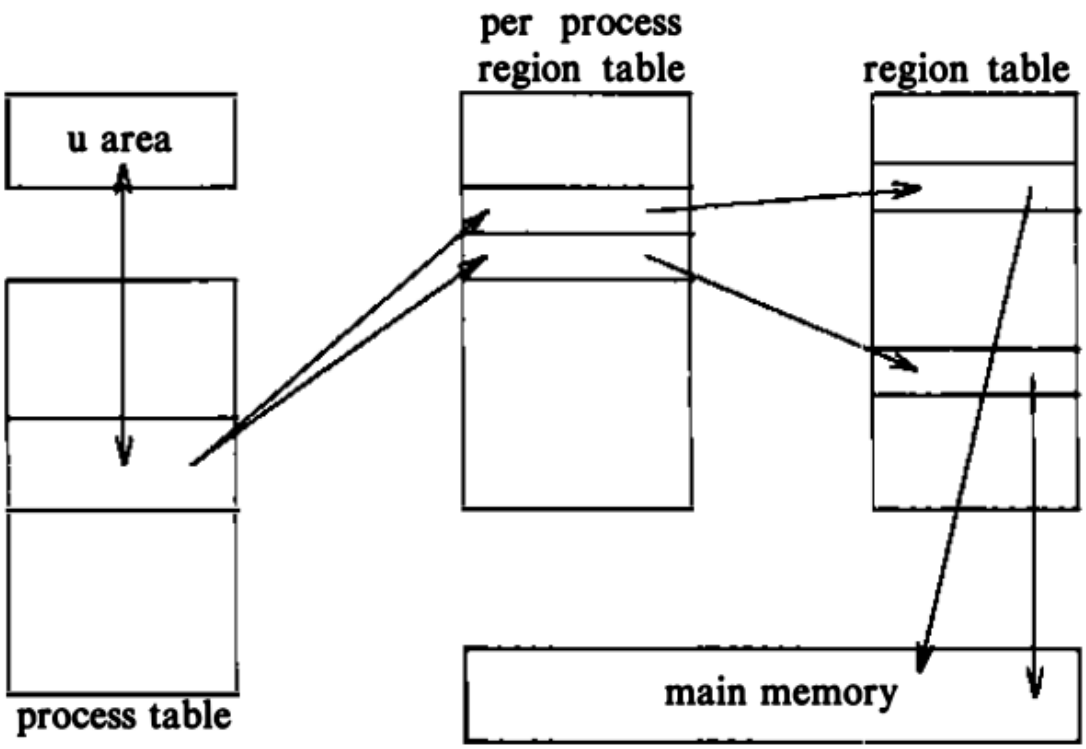
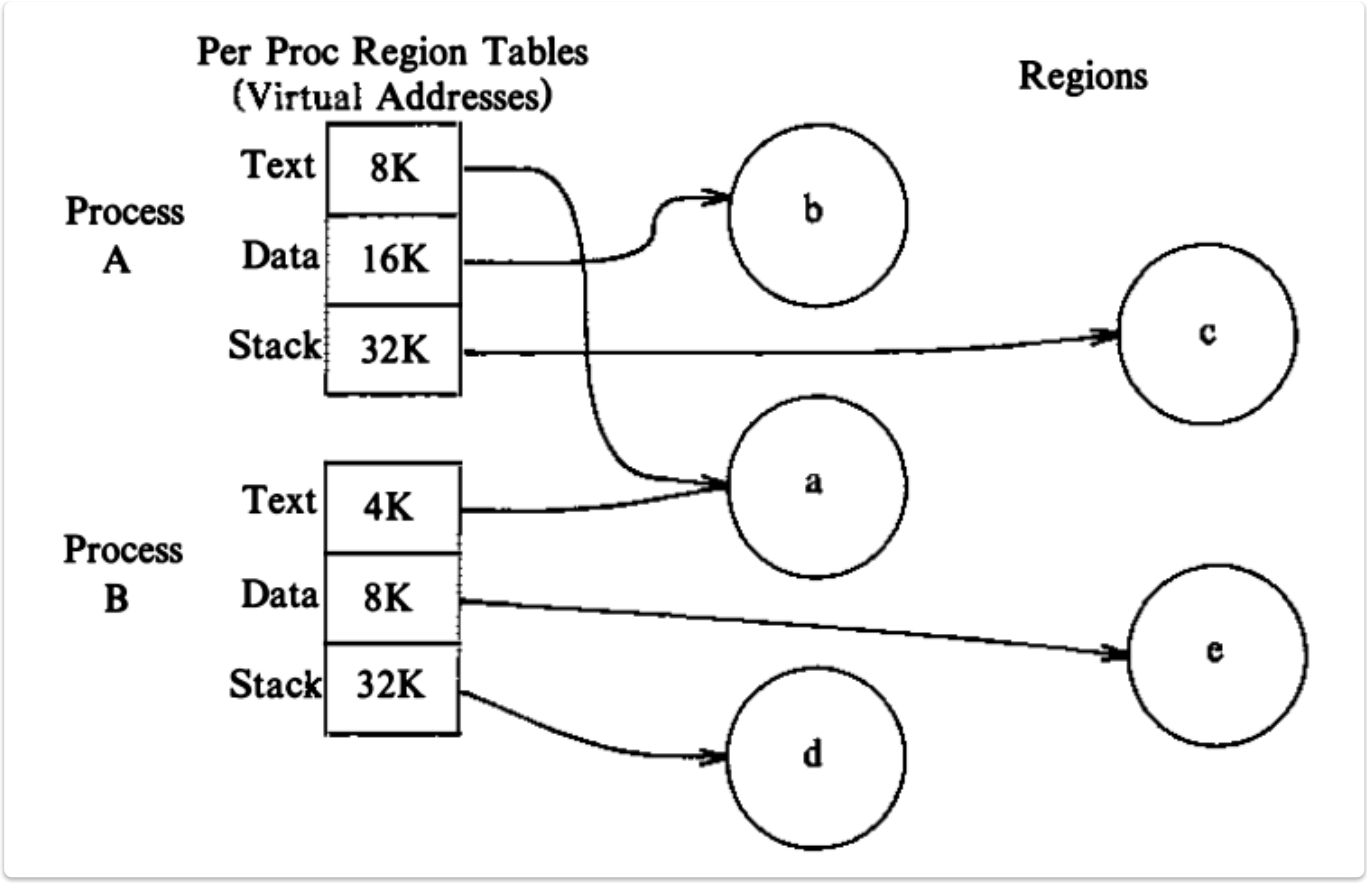


Figure 2.5. Data Structures for Processes

Regions

- It is a continuous area of virtual address space of a process that can be treated as an distinct object to be shared or protected.
- Regions: **Text**, **Data**, and **Stack**.
- **Several process can share a region**. Example: several process may execute the same program, so they can share the text region. Similarly, they may share a common shared-memory region.
- The kernel allocates an entry for region table for each active region in the system.
- Each process contain a private **per process region table**, called a **pregion**.
- Region table contains the information to determine where a region's contents are located in physical
  memory.
- pregion entries can exist in process table, u area, or a seperately allocated region.
- Each pregion entry points to a region table entry.
- Several processes can share parts of their address space via a region
- Each process accesses the region via a private pregion entry.
- The concept of regions is independent of the memory management policies implemented by the OS. (swapping, paging, etc)

Two Processes sharing a text region: 8K and 4K map to the same location (same physical address)



Pages and Page Tables

- The memory management hardware divides the physical memory into a set of equal-sized blocks called **pages**.
- Page Size ranges from 512 bytes to 4K bytes.
- Every memory location can be addressed by a

$$(page\ number,\ byte\ offset\ in\ page)$$

- For example: Physical memory size $= 2^{32}$, page size $= 1K\ bytes = 1024 = 2^{10}\ bytes$, then it has $\frac{2^{32}}{2^{10}} = 2^{32-10} = 2^{22}$ pages, that equals a 22 bit page number and a 10 bit byte offset.
- When a kernel assigns physical pages of memory to a region, it may not assign them in a particular order or continuously.

Hexadecimal Address	58432	
Binary	0101 1000 0100 0011 0010	
Page Number, Page Offset	01 0110 0001	00 0011 0010
In Hexadecimal	161	32

Figure 6.3. Addressing Physical Memory as Pages

Logical Page Number	Physical Page Number
0	177
1	54
2	209
3	17

Figure 6.4. Mapping of Logical to Physical Page Numbers

Example

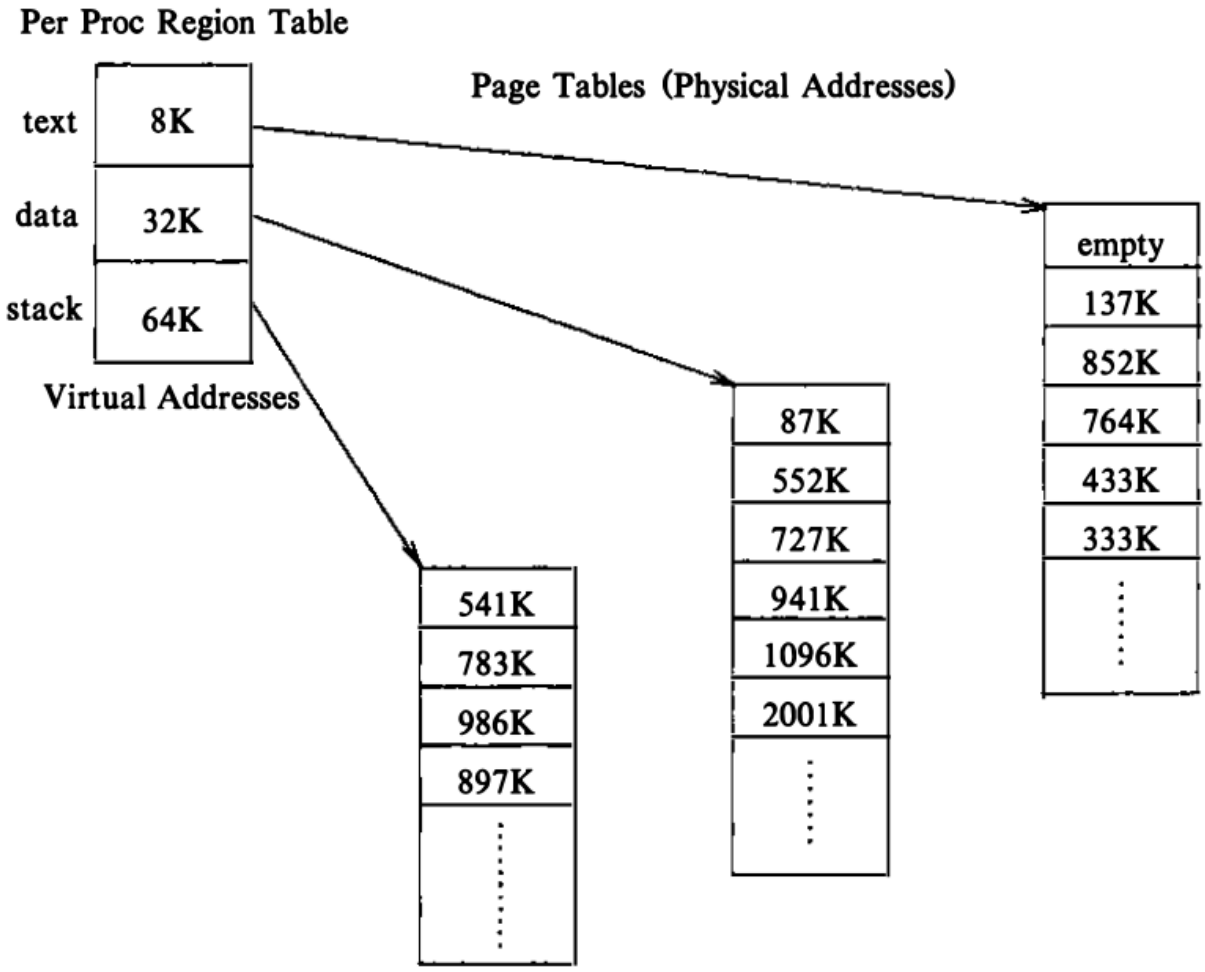


Figure 6.5. Mapping Virtual Addresses to Physical Addresses

In the above,

- text region **starts** at 8K
- data region starts at 32K
- stack region starts at 64K

If we want to access address 68,432

so offset in stack region $= 68432 - 64 \times 1024 = 2896$

So, page number $= \lfloor 2896/1024 \rfloor = 2$
and byte offset $= 2896 \mod 1024 = 848$

So address is located at $986K$ in physical memory at byte offset 848. The memory location $68,432$ is in page that is at $986K$ and that location is in the offset 848.