Processes in Linux:
An Overview of Linux Process Management

Dr. Vimal Baghel

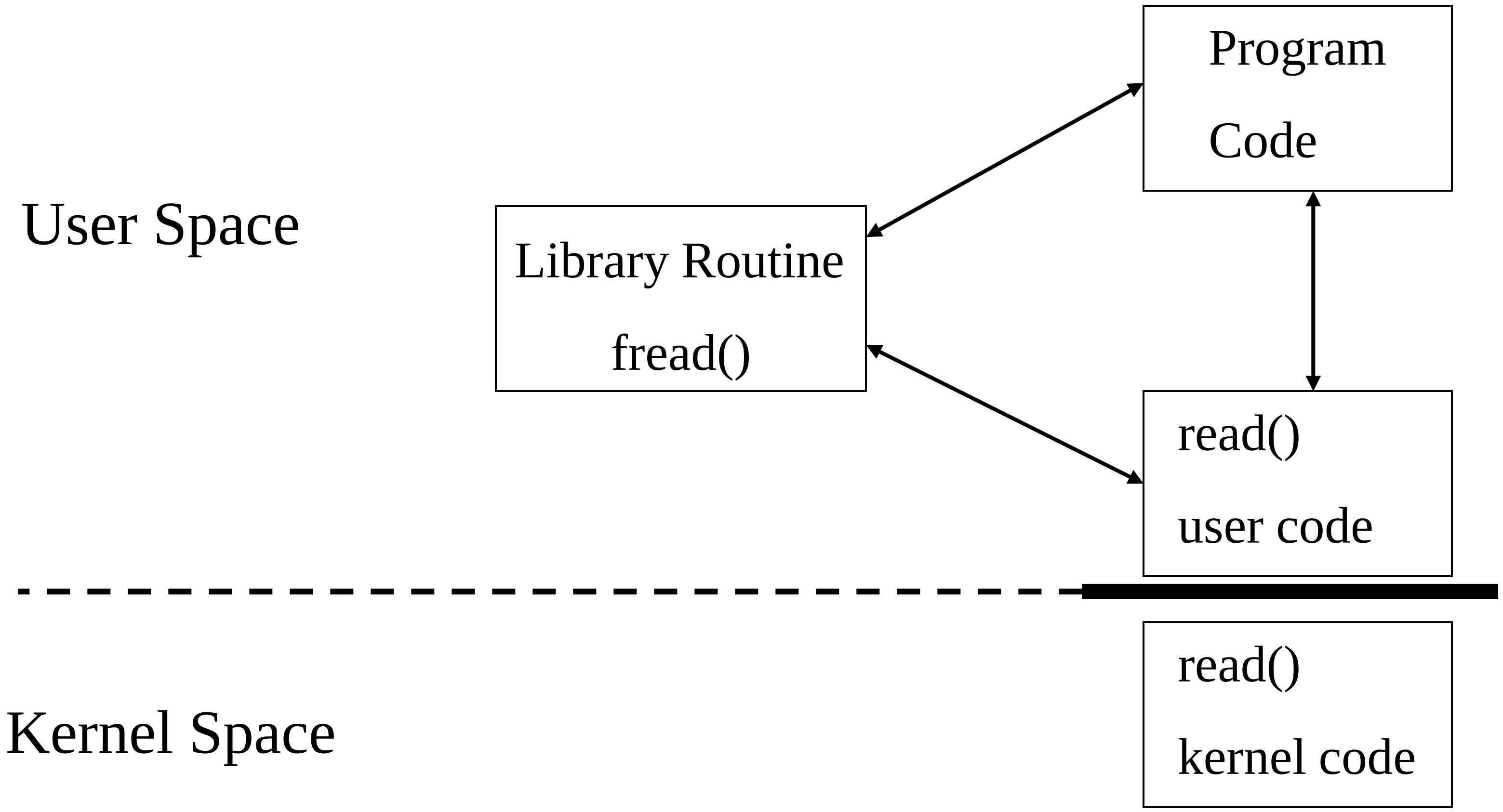
Assistant Professor

SCSET, BU

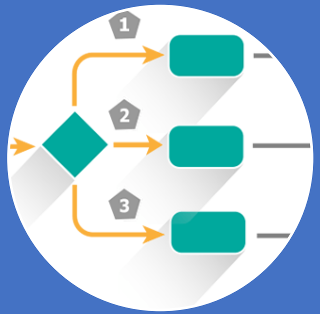
Outline

- Introduction to Linux Processes
- Processes Resources
- Q&A

User and System Space



What is a Process?:



A process is a program in execution state.

It is a dynamic Entity in OS.

The Process

The process includes

- the PC,
- CPU's registers,
- the process stacks containing temporary data such as routine parameters, return addresses and saved variables.

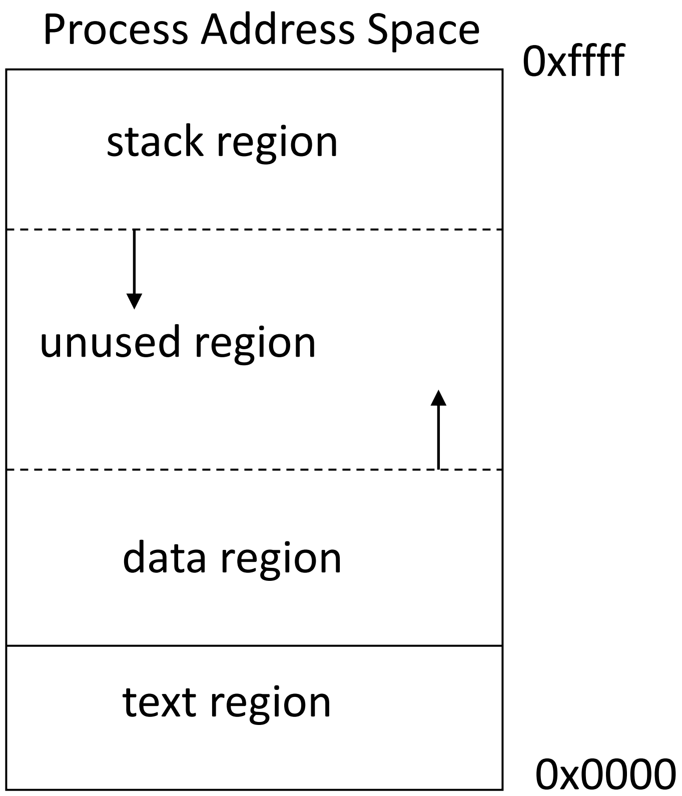
Process Description

- A process is completely defined by
  - the CPU registers
    - program counter, stack pointer, control, general purpose, etc.
  - memory regions
    - user and kernel stacks
    - code
    - heap
- To start and stop a program, all of the above must be saved or restored
  - CPU registers must be explicitly saved/restored
  - memory regions are implicitly saved/restored

Memory Regions of a Process

- Every process has 3 main regions
  - text area
    - stores the actual program code
    - static in size (usually)
  - stack area
    - stores local data
      - function parameters, local variables, return address
  - data area (heap)
    - stores program data not on the stack
    - grows dynamically per user requests

Memory Regions of a Process



Note: the stack usually grows down while the data region grows upward – the area in between is free

User vs. Kernel Stack

- Each process gets its own user stack
  - resides in user space
  - manipulated by the process itself
- In Linux, each process gets its own kernel stack
  - resides in kernel space
  - manipulated by the OS
  - used by the OS to handle system calls and interrupts that occur while the process is running

User Stack

Function: *printAvg*
Return: *check* call inst
Param: avg
Local: none

Function: *check*
Return: *main* call inst
Param: grade
Local: hi, low, avg

Method: *main*
Return: halt
Param: command line
Local: grade[5], num

Kernel Stack

Function: *calcSector*
Return: *read* call inst
Param: avg
Local: sector

Function: *read*
Return: *user program*
Param: block
Local: sector

User program counter
User stack pointer

Process Descriptor

- OS data structure that holds all necessary information for a process
  - process state
  - CPU registers
  - memory regions
  - pointers for lists (queues)
  - etc.

Process Descriptor

<table>
<tr><td>pointer</td><td>state</td></tr>
<tr><td colspan="2">process ID number</td></tr>
<tr><td colspan="2">program counter</td></tr>
<tr><td colspan="2">registers</td></tr>
<tr><td colspan="2">memory regions</td></tr>
<tr><td colspan="2">list of open files</td></tr>
<tr><td colspan="2">.
.
.</td></tr>
</table>

Process Descriptor

- Pointer
  - used to maintain queues that are linked lists
- State
  - current state the process is in (i.e. running)
- Process ID Number
  - identifies the current process
- Program Counter
  - needed to restart a process from where it was interrupted

Process Descriptor

- Registers
  - completely define state of process on a CPU
- Memory Limits
  - define the range of legal addresses for a process
- List of Open Files
  - pretty self explanatory

Background & Foreground Processes

- A foreground process is any process which is not continuously running and it waiting on something like user input
- A background process is something that is continually running and does not require any additional input
- Can someone name examples of each?

Moving a Process to the Background

- When executing commands on the command line, there is usually some output that is displayed on the terminal
- If you move a process to the background, the output will not be shown

Background Process Example

- Usually, when you download a file from the command line, the status is displayed on the terminal
- To move a process to the background all we will do is add an ampersand (&) at the end of the command
  - **wget http://releases.ubuntu.com/24.04.2/ubuntu-24.04.2-desktop-amd64.iso_ga=2.142658160.410030815.1551071806-1676866732.1550780350 &**
- Now this will be moved to the background

Moving back to the Foreground

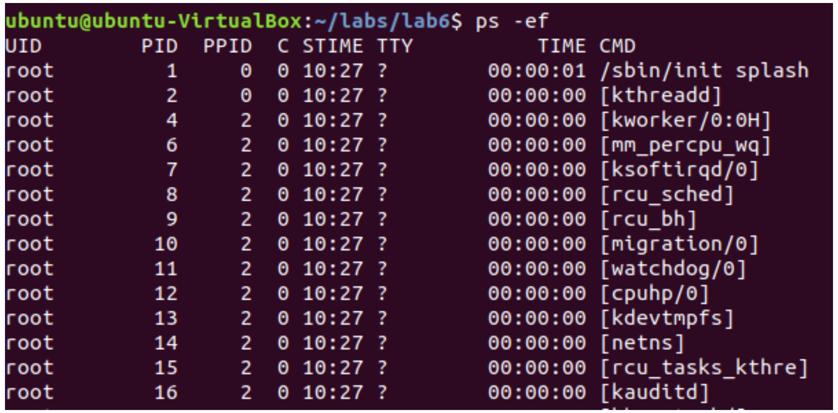
- To move a process back to the foreground, use the following steps:
  - Use the **jobs** command to identify the job number of the background process
  - Then use the **fg** command to bring it back with the following syntax
    - **fg [job number]**

Different Types of Processes

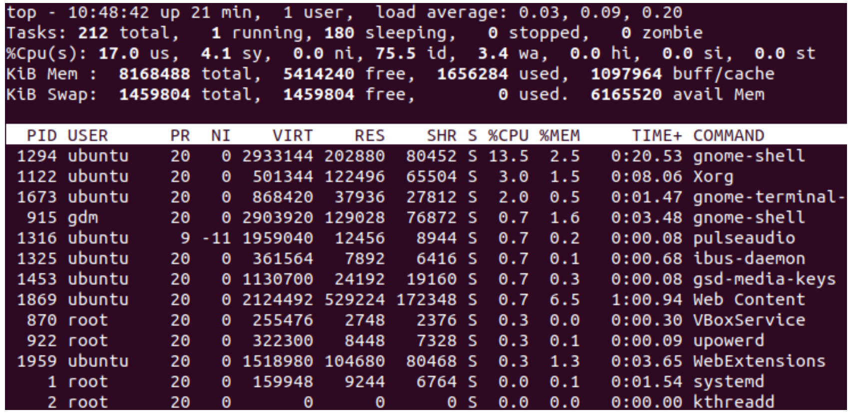
- There are four types of processes:
  - Running: current process that is being executed in the operating system
  - Waiting: process which is waiting for system resources to run
  - Stopped: process that is not running
  - Zombie: process whose parent processes has ended, but the child process is still in the process table

Viewing Processes

- Two commands you can use to view the process from the command line: **ps** and **top**
- To view all the processes with **ps,** use **ps -ef**



ps -ef



top

Ending a Process In Linux

- Sometimes we need to end a program or process from the command line.
- Use the following steps:
  1. Locate the process id [PID] of the process/program you want to kill
  2. Use the **kill** command with the following syntax: **kill [PID]**
  3. If the process is still running, do the following: **kill -9 [PID]**
  4. The -9 is a SIGKILL signal telling the process to terminate immediately

Ending All Process

- We can use the **killall** command to kill multiple processes at the same time
- Syntax: **killall [options] PIDs**
- Or you can use **pkill –u [username]** to kill all processes started by [username]

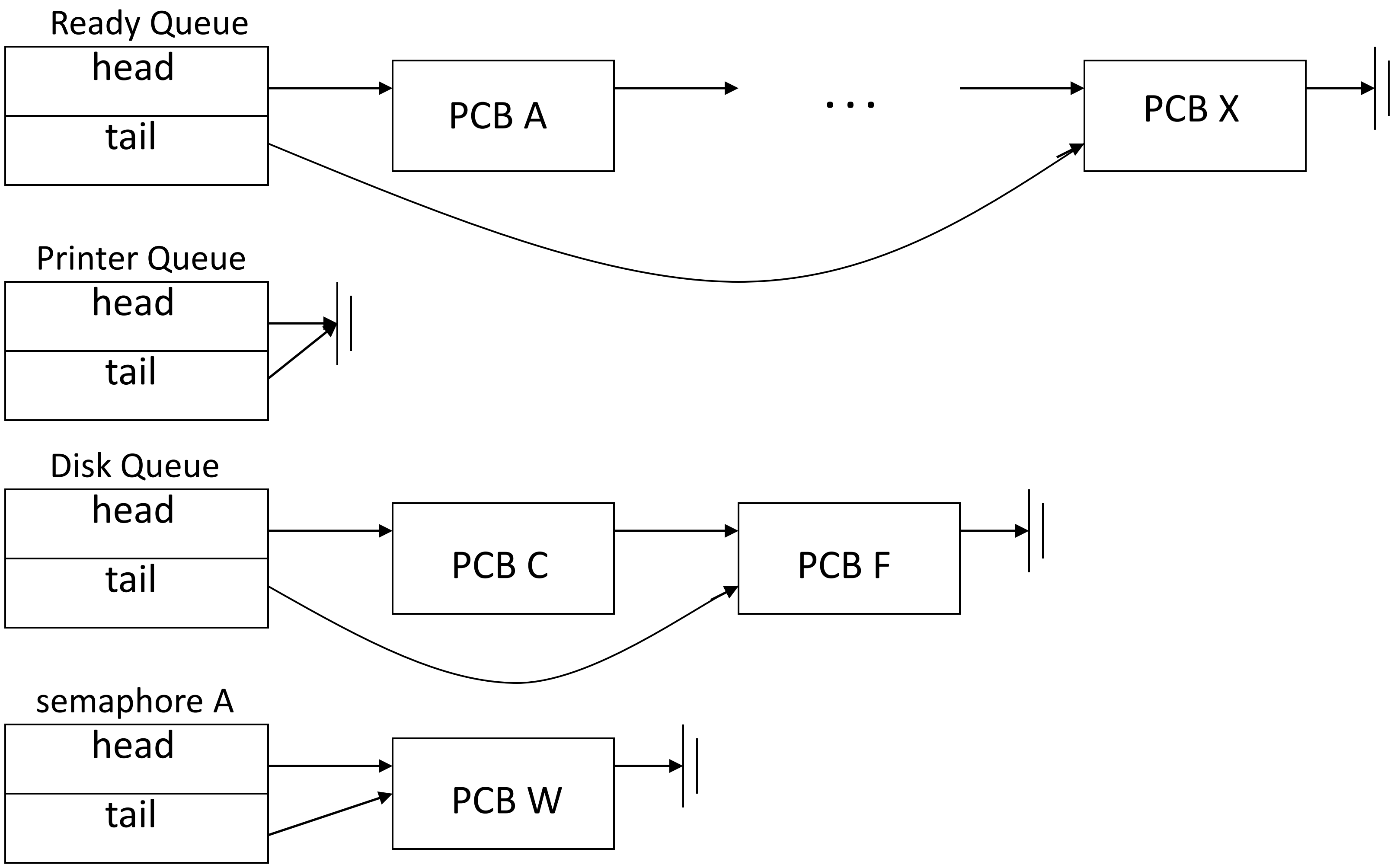
Process States

- 5 generic states for processes
  - new
  - ready
  - running
  - waiting
  - terminated (zombie)
- Many OS's combine ready and running into *runnable* state

Process Queues

- Every process belongs to some queue
  - implemented as linked list
  - use the pointer field in the process descriptor
- Ready queue
  - list of jobs that are ready to run
- Waiting queues
  - any job that is not ready to run is waiting on some event
    - I/O, semaphores, communication, etc.
  - each of these events gets its own queue

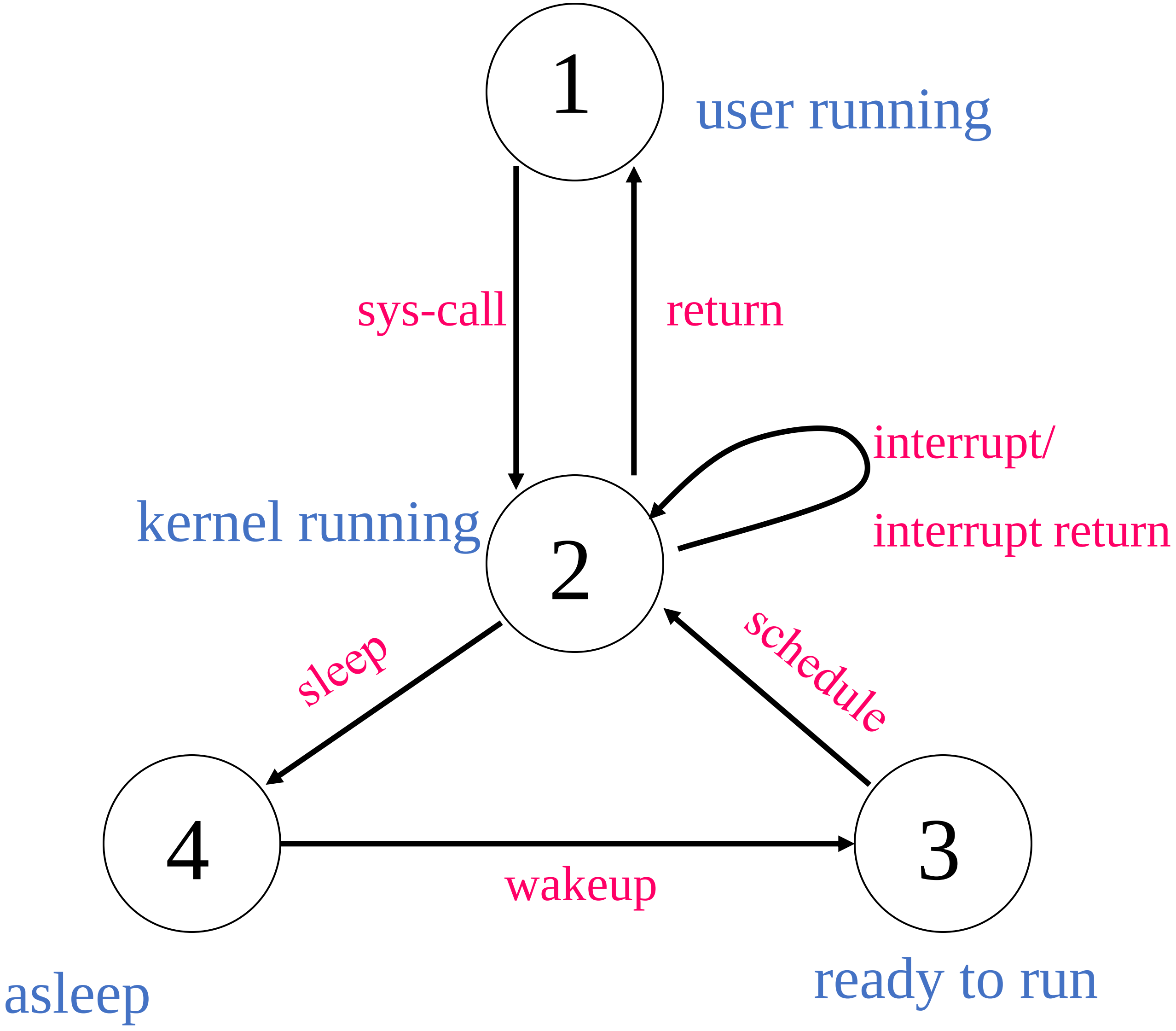
Process Queues



Process States Transitions

- A process changes *state* according to its circumstances
  - Running or ready
  - Waiting for an event or for a resource, (Interruptible or Uninterruptible)
  - Stopped by receiving a signal,
  - zombie – a halted process

Process State Transitions



How do Processes Actually Work?

- In the Linux, processes are created by a method called ***"forking"***
  - Forking is done when the OS duplicated a process
  - The original process is called the parent process
  - And the new process is the child process

Forkbombing Example

- What is forkbombing?
  - It is when you spawn multiple processes which leads to lack of system resources and other very bad things
- I will forkbomb the server to demonstrate the **pkill** command
- DO NOT TRY THIS (the Systems department will not be happy with you)

Creating Processes

- Parent process creates a child process
  - results in a *tree*
- Execution options
  - parent and child execute concurrently
  - parent waits for child to terminate
- Address space options
  - child gets its own memory
  - child gets a subset of parents memory

Creating Processes in Unix

```
void main() {
  int pid;
  pid = fork();
  if(pid == 0) { // child process - start a new program
      execlp("/bin/ls", "/home/mattmcc/", NULL);
  }
  else {          // parent process - wait for child
      wait(NULL);
      exit(0);
  }
}
```

Creating Processes in Unix

- *fork()* system call
  - creates **exact** copy of parent
  - only thing different is return address
    - child gets 0
    - parent gets child ID
  - child may be a *heavyweight process*
    - has its own address space
    - runs concurrently with parent
  - child may be a *lightweight* process
    - shares address space with parent (and siblings)
    - still has its own execution context and runs concurrently with parent

Creating Processes in Linux

- *exec()* system call starts new program
  - needed to get child to do something new
  - remember, child is exact copy of parent
- *wait()* system call forces parent to suspend until child completes
- *exit()* system call terminates a process
  - places it into zombie state

Destroying a Process

- Multiple ways for a process to get destroyed
  - process issues and *exit()* call
  - parent process issues a *kill()* call
  - process receives a terminate signal
    - did something illegal
- On death:
  - reclaim all of process's memory regions
  - make process unrunnable
  - put the process in the *zombie state*
  - ***However, do not remove its process descriptor from the list of processes***

Zombie State

- Why keep process descriptor around?
  - parent may be waiting for child to terminate
    - via the *wait()* system call
  - parent needs to get the exit code of the child
    - this information is stored in the descriptor
  - if descriptor was destroyed immediately, this information could not be retrieved
  - after getting this information, the process descriptor can be removed
    - no more remnants of the process

init Process

- This is one of the first processes spawned by the OS
  - is an ancestor to all other processes
- Runs in the background and does clean-up
  - looks for zombie’s whose parents have not issued a *wait()*
    - removes them from the system
  - looks for processes whose parents have died
    - adopts them as its own

Process Representation: task_struct

- Process resources: `$ ls -l /proc/self/`
- Each Linux process is represented by a data structure: ***task_struct (A PCB/TCB in Linux)***

- The task vector is an array of pointers to every ***task_struct*** in the system

- The max number of processes are limited by the size of the task vector, default is 512.

Process Representation: task_struct

- As the new processes are created, a new ***task_struct*** is allocated from system memory and added into the ***task*** vector.

- To make it easy to find, the current running process is pointed to by the *current* pointer

- Linux also supports real time process.

Processes and its Resources

- A process is an OS abstraction that groups together multiple resources:

  - ✓ An address space
  - ✓ One or more threads
  - ✓ Opened files
  - ✓ Sockets
  - ✓ Semaphores
  - ✓ Shared memory regions
  - ✓ Timers
  - ✓ Signal handlers
  - ✓ Many other resources and status information

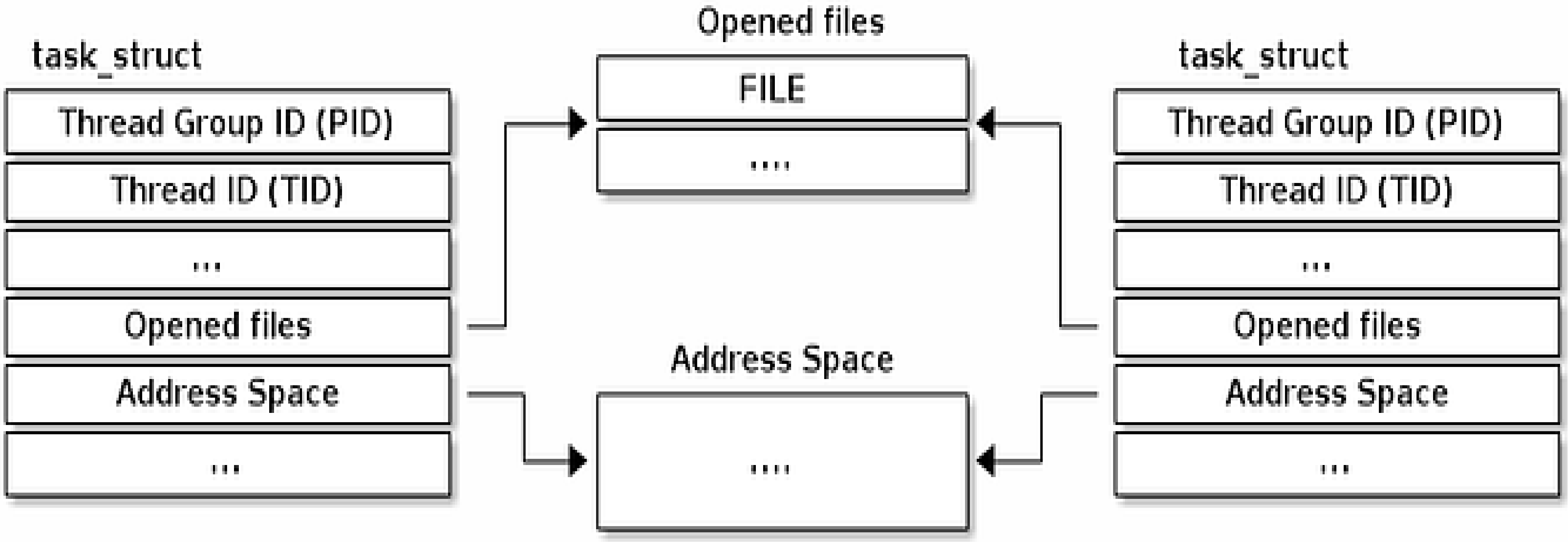
All this information is grouped in the Process Control Group (PCB). In Linux this is **struct task_struct**.

Overview of process resources

- A summary of the resources a process has can be obtained from the */proc/<pid>* directory, where *<pid>* is the process id for the process we want to look at.

- $ ls -1 /proc/pid
- $ ls -1 /proc/self

2 processes opening the same file and the relationship between the 2 different "task_struct"



Process representation in Linux

- Process is represented by a large structure **task_struct**.
- It contains:
  - The necessary data to represent the process
  - data for accounting and to maintain relationships with other processes (parents and children)

Managing the task Array

- The **task** array is updated every time a process is created or destroyed.
- A separate list (headed by **tarray_freelist**) keeps track of free elements in the **task** array.
  - When a process is destroyed its entry in the **task** array is added to the head of the freelist.

task_struct

- Although, **task_struct** is quite large and complex data structure, but its fields can be divided into several functional areas:
  - **State**
  - **Scheduling Information**
  - **Identifiers**
  - **IPC**
  - **Links**
  - **Times and Timers**
  - **File system**
  - **Virtual memory**
  - **Processor Specific Context**

Scheduling Info

- The scheduler needs this information in order to fairly decide which process in the system most deserves to run.

Process Identifiers

- Every process in the system has a process identifier (PID).
- The PID is not an index into the task vector, it is simply a number.
- Each process also has User and group identifiers, these are used to control this processes access to the files and devices in the system,

IPC

- Linux supports the classic Unix IPC mechanisms:
  - signals,
  - pipes
  - semaphores
  - the System V IPC mechanisms of shared memory, semaphores and message queues.

Process Links

- No process in Linux is independent
- Every process in the system, except the initial process has a parent process.
- New processes are not created, **they are copied, or rather *cloned* from previous processes.**
- Every *task_struct* representing a process keeps pointers to its parent process and to its siblings (those processes with the same parent process) as well as to its own child processes.

Process Links

- You can see the family relationship between the running processes in a Linux system using
  the pstree command:

```
init(1)-+-crond(98)
        |-emacs(387)
        |-gpm(146)
        |-inetd(110)
        |-kerneld(18)
        |-kflushd(2)
        |-klogd(87)
        |-kswapd(3)
        |-login(160)---bash(192)---emacs(225)
        |-lpd(121)
        |-mingetty(161)
        |-mingetty(162)
        |-mingetty(163)
        |-mingetty(164)
        |-login(403)---bash(404)---pstree(594)
        |-sendmail(134)
        |-syslogd(78)
        `-update(166)
```

Process Links

- Additionally, all the processes in the system are held in a doubly linked list whose root is the init processes task_struct data structure.
- This list allows the Linux kernel to look at every process in the system.
- It needs to do this to provide support for commands such as ps or kill.

Times and Timers

- The kernel keeps track of a processes creation time as well as the CPU time that it consumes during its lifetime.
- Each clock tick, the kernel updates the amount of time in jiffies that the current process has spent in system and in user mode. Linux also supports process specific *interval* timers, processes can use system calls to set up timers to send signals to themselves when the timers expire.
- These timers can be single-shot or periodic timers.

File system

- Processes can open and close files as they wish and the processes task_struct contains pointers to descriptors for each open file as well as pointers to two VFS inodes.
- Each VFS inode uniquely describes a file or directory within a file system and also provides a uniform interface to the underlying file systems.
- The first is to the root of the process (its home directory) and the second is to its current or *pwd* directory.
- These two VFS inodes have their count fields incremented to show that one or more processes are referencing them.
- This is why you cannot delete the directory that a process has as its *pwd* directory set to, or for that matter one of its sub-directories.

Virtual memory

- Most processes have some virtual memory (kernel threads and daemons do not) and the Linux kernel must track how that virtual memory is mapped onto the system's physical memory.

Processor Specific Context

- A process could be thought of as the sum total of the system's current state.
- Whenever a process is running it is using the processor's registers, stacks and so on. This is the processes context and, when a process is suspended, all of that CPU specific context must be saved in the **task_struct** for the process.
- When a process is restarted by the scheduler its context is restored from here.

Thanks

Q & A