CSE308 Operating Systems

Operations on Processes

Dr S.Rajarajan

SoC

SASTRA

Process Creation and termination

- The processes in most systems can execute concurrently, and they may be **created** and **deleted** dynamically.
- Thus, these systems must provide a mechanism for **process creation and termination.**
- We explore the mechanisms involved in creating processes

Process Creation

- During the course of execution, a process may create (**spawn**) several new processes.
- The creating process is called a **parent process**, and the new processes are called the **children** of that process.
- Each of these new processes may in turn create other processes, forming **a tree** of processes.
- Most operating systems identify processes according to a unique **process identifier (or pid**), which is typically an integer number.
- The pid provides **a unique value** for each process in the system, and it can be used as an **index** to access various attributes of a process **within the kernel**.

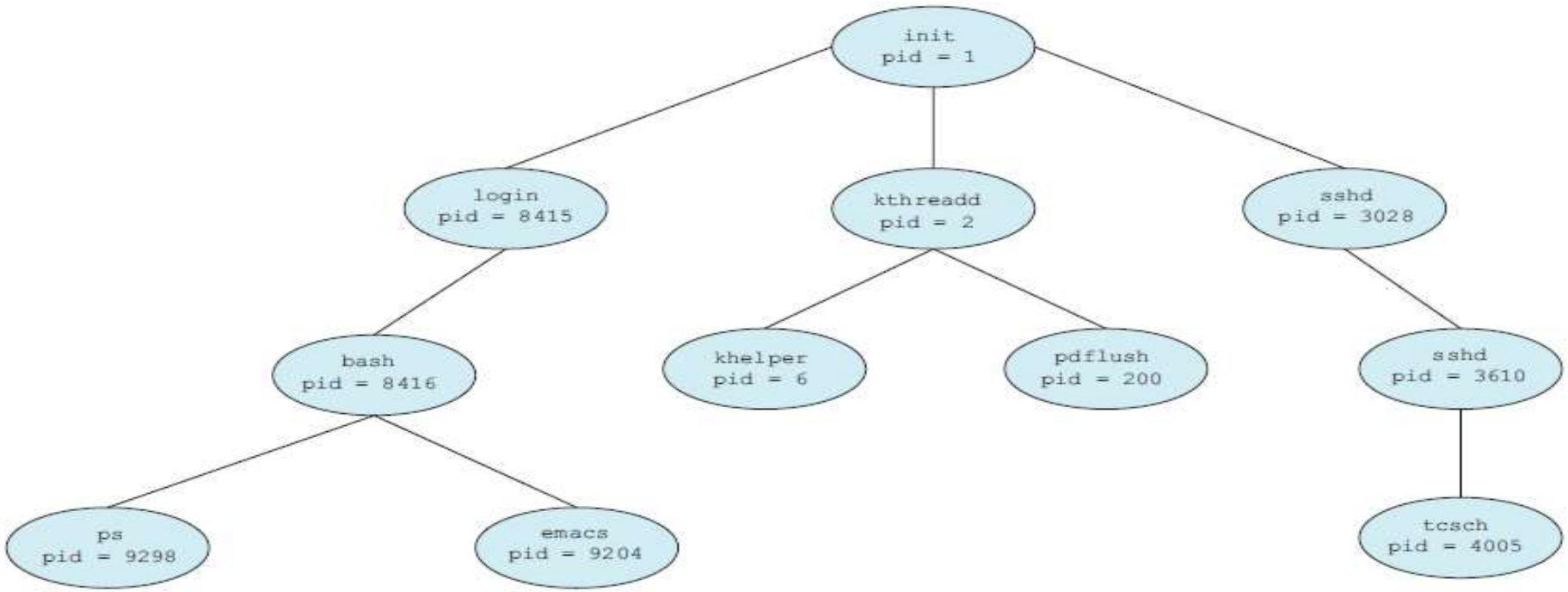

Figure 3.8 A tree of processes on a typical Linux system.

- The **init process** (which always has a **pid of 1**) serves as the **root parent process** for all user processes.
- Once the system has booted, the init process can also create various user processes.
- In Figure we see three **children of init : login, kthreadd** and **sshd**.
  - **kthreadd** process is responsible for creating additional processes that perform tasks on behalf of the kernel.
  - **sshd** process is responsible for managing clients that connect to the system by using ssh (which is short for ***secure shell).***
  - **login process** is responsible for managing clients that directly log onto the system.
- In this example, **a client** has logged on and is using the **bash shell**, which has been assigned **pid 8416**.
- Using the bash command-line interface, this user has created the process **ps** as well as the **emacs editor**

- On UNIX and Linux systems, we can obtain **a listing of processes** by using the ps command. For example, the command
  - **ps –el**
- will **list complete information for all processes** currently active in the system by recursively tracing parent processes all the way to the init process

Child Process

- When a process creates a child process, that child process will need certain resources (CPU time, memory, files, I/O devices) to accomplish its task.
- A **child process** may be able to **obtain its resources** directly from the **operating system**, or it may be constrained to a **subset of the resources** of the **parent process**.
- The parent may have to **partition its resources** among its children, or it may be able to share some resources (such as **memory or files**) among several of its **children.**
- Restricting a child process to a subset of the parent's resources prevents any process **from overloading the system** by creating **too many child processes.**

- In addition to supplying various **physica**l and **logical resources**, the parent process may pass along **initialization data** (**input**) to the child process.
- When a process creates a new process, two possibilities for execution exist:
  - The **parent continues to execute** concurrently with its children
  - The **parent waits** until some or all of its children have terminated
- There are also two address-space possibilities for the new process
  - The child process is a **duplicate of the parent** process (it has the same program and data as the parent).
  - The child process has a **new program loaded** into it.

Fork

- A new process is created by the fork() system call.
- The new process consists of **a copy of the address space** of the parent process.
- This mechanism allows the parent process to **communicate easily** with its child process.
- Both processes (the parent and the child) continue execution at the instruction **after the fork(),**
- The return code for the **fork() is zero** for the new (**child**) process, whereas the (**nonzero**) process identifier of the child is returned to the parent
- Because the child is a copy of the parent, each process has its **own copy of any data**

- After a fork() system call, one of the two processes typically uses the **exec() system c**all to replace the process's memory space with a new program.
- The exec() system call **loads a binary file** into memory (destroying the memory image of the program containing the exec() system call) and starts its execution.
- The parent can then create more children; or, if it has nothing else to do while the child runs, it can issue a **wait() system call** to move itself off the ready queue until the termination of the child.

```
pid = fork();

if (pid < 0) { /* error occurred */
   fprintf(stderr, "Fork Failed");
   return 1;
}
else if (pid == 0) { /* child process */
   execlp("/bin/ls","ls",NULL);
}
else { /* parent process */
   /* parent will wait for the child to complete */
   wait(NULL);
   printf("Child Complete");
}
```

execlp() is a version of the exec() system call)

- We now have **two** different processes **running copies of the same program.**
- The only difference is that the **value of pid** (the process identifier) for the child process is zero, while that for the parent is an integer value greater than zero (in fact, it is the actual pid of the child process).
- The **child process inherits privileges** and **scheduling attributes** from the parent, as well certain resources, such as **open files**.
- The child process then **overlays its address space** with the UNIX command **/bin/ls** (used to get a directory listing) using the **execlp()** system call (execlp() is a version of the exec() system call).
- The **parent waits for the child** process to complete with the wait() system call.
- When the child process completes (by either implicitly or explicitly invoking exit()), the parent process resumes from the call to wait(), where it completes using the exit() system call.

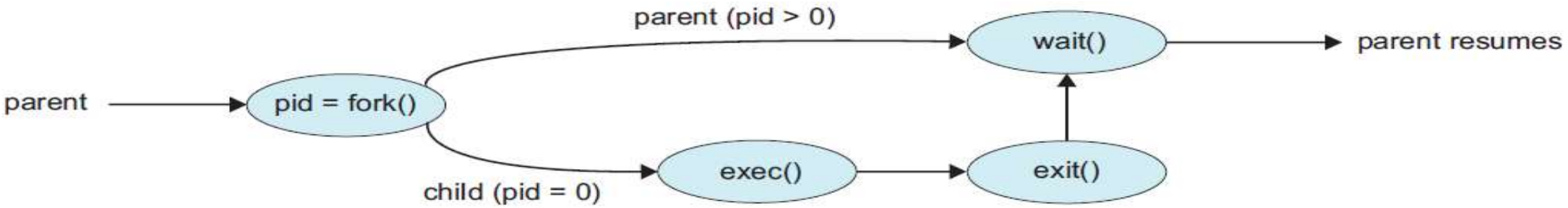

Figure 3.10 Process creation using the `fork()` system call.

- Because the call to exec() overlays the process's address space with a new program, the call to exec() does not return control unless an error occurs

CreateProcess() Vs fork()

- Processes are created in the **Windows API** using the CreateProcess() function, which is similar to fork() in that a parent creates a new child process.
- whereas **fork()** has the child **process inheriting the address space of its parent**, CreateProcess() requires **loading a specified program into the address space** of the child process at process creation.
- fork() is passed **no parameters**, CreateProcess() expects no **fewer than ten parameters**.

```
/* create child process */
if (!CreateProcess(NULL, /* use command line */
 "C:\\WINDOWS\\system32\\mspaint.exe", /* command */
 NULL, /* don't inherit process handle */
 NULL, /* don't inherit thread handle */
 FALSE, /* disable handle inheritance */
 0, /* no creation flags */
 NULL, /* use parent's environment block */
 NULL, /* use parent's existing directory */
 &si,
 &pi))
{
```

Process Termination

- A process terminates when it **finishes executing** its final statement and asks the operating system to delete it by using the **exit() system ca**ll.
- At that point, the process **may return a status value** (typically an integer) **to its parent** process (via the **wait() system call**).
- All the **resources** of the process—including physical and virtual memory, open files, and I/O buffers—are **de-allocated** by the operating system.

- Termination can **occur in other circumstances** as well.
- A process can cause the **termination of another process** via an appropriate **system call** (for example, **TerminateProcess()** in Windows).
- Usually, such a system call **can be invoked only** by the **parent** of the process that is to be terminated.
- Otherwise, users **could arbitrarily kill** each other's jobs.
- Note that a parent needs to know the **identities of its children** if it is **to terminate them.**
- Thus, when one process creates a new process, the identity of the newly created process is passed to the parent (**fork return value**)

Parent killing child

- A parent may terminate the execution of one of its children for a variety of reasons, such as these:
  - The child has **exceeded its usage** of some of the resources that it has been allocated.
  - The **task assigned** to the child is **no longer required**.
  - The **parent is exiting**, and the operating system does not allow a child to continue if its parent terminates. This phenomenon, referred to as **cascading termination**, is normally initiated by the operating system.

- We can terminate a process by using the **exit() system call.**
- A **parent process** may **wait for the termination** of a child process by using the **wait() system call**.
- The wait() system call is **passed a parameter** that allows the parent to obtain the **exit status** of the child.
- This system call also returns the **process identifier** of the terminated child so that the parent can tell which of its children has terminated:

```
pid_t pid;
int status;

pid = wait(&status);
```

- When a child process terminates, its **resources are deallocated** by the operating system.
- However, its **entry in the process table** must remain there until the **parent calls wait(),** because the process table contains the process's **exit status.**
- A process that has terminated, but whose **parent** has **not yet called wait(),** is known as a **zombie process**
- Once the parent **calls wait(),** the process identifier of the **zombie process** and its entry in the **process table are released.**

- Consider what would happen if a **parent did not invoke wait()** and instead terminated, thereby leaving its child processes as **orphans.**
- Linux and UNIX address this scenario by assigning the **init process** as the new parent to orphan processes.
- The **init** process **periodically invokes wait(),** thereby allowing the **exit status of any orphaned process** to be collected and **releasing the orphan's** process identifier and process-table entry