Operations on Processes

- System must provide mechanisms for:
    - process creation,
   - process termination,

Process Creation

- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes

- Generally, process identified and managed via a **process identifier** (**pid**)

- Resource sharing options
  - Parent and children share all resources

- Children share subset of parent's resources
- Parent and child share no resources
- Execution options
  - Parent and children execute concurrently
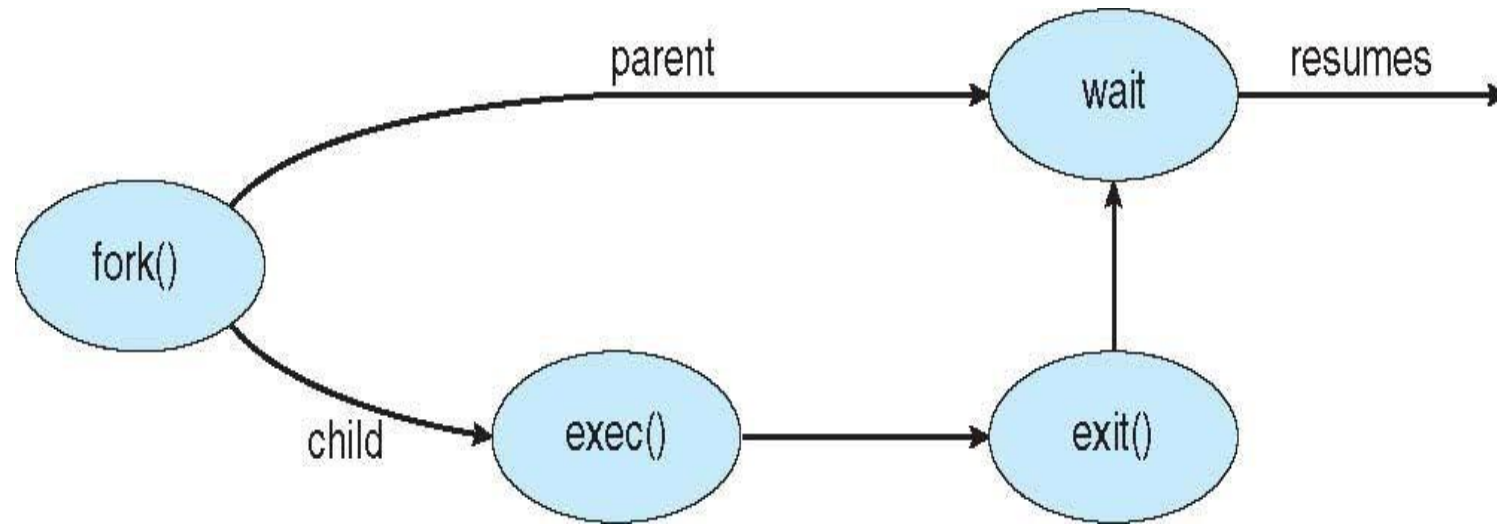 - Parent waits until children terminate

A Tree of Processes in Linux
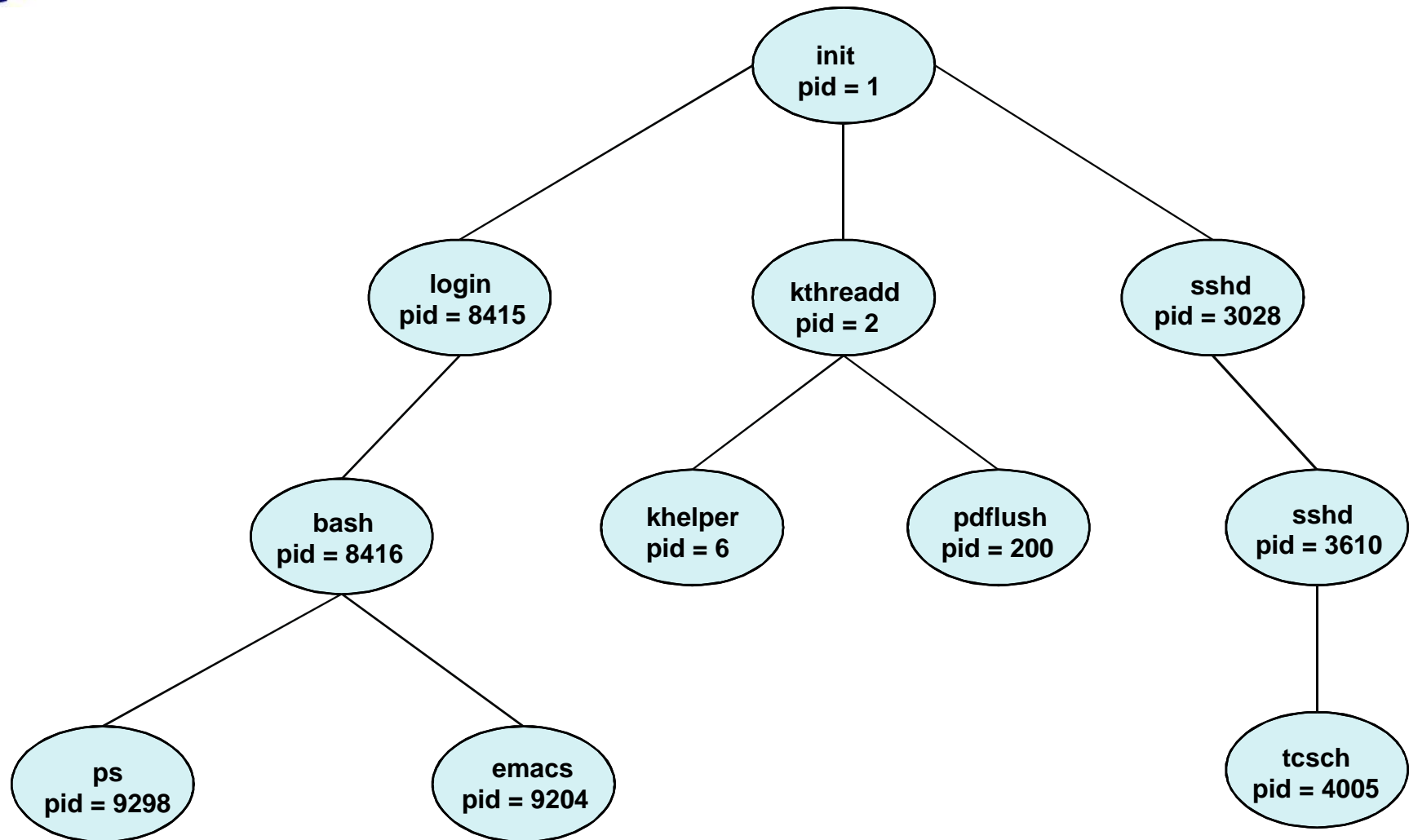
Process Creation (Cont.)

- Address space
    - Child duplicate of parent
    - Child has a program loaded into it
- UNIX examples
    - `fork()` system call creates new process
    - `exec()` system call used after a `fork()` to replace the process' memory space with a new program

C Program Forking Separate Process

```c
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    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");
    }

    return 0;
}
```

Creating a Separate Process via Windows API

```c
#include <stdio.h>
#include <windows.h>

int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

    /* allocate memory */
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    /* 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))
    {
       fprintf(stderr, "Create Process Failed");
       return -1;
    }
    /* parent will wait for the child to complete */
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");

    /* close handles */
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

Process Termination

- Process executes last statement and then asks the operating system to delete it using the `exit()` system call.
  - Returns status data from child to parent (via `wait()`)
  - Process' resources are deallocated by operating system
- Parent may terminate the execution of children processes using the `abort()` system call. Some reasons for doing so:
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates

Process Termination

- Some operating systems do not allow child to exists if its parent has terminated. If a process terminates, then all its children must also be terminated.

  - **cascading termination.** All children, grandchildren, etc. are terminated.

  - The termination is initiated by the operating system.

- The parent process may wait for termination of a child process by using the `wait()` system call`.`   The call returns status information and the pid of the terminated process

  ```
  pid = wait(&status);
  ```

- If no parent waiting (did not invoke `wait()`) process is a **zombie**

- If parent terminated without invoking `wait` , process is an **orphan**

Multiprocess Architecture – Chrome Browser

- Many web browsers ran as single process (some still do)
  - If one web site causes trouble, entire browser can hang or crash
- Google Chrome Browser is multiprocess with 3 different types of processes:
  - **Browser** process manages user interface, disk and network I/O
  - **Renderer** process renders web pages, deals with HTML, Javascript. A new renderer created for each website opened
- Runs in **sandbox** restricting disk and network I/O, minimizing effect of security exploits
  - **Plug-in** process for each type of plug-i

Each tab represents a separate process