LINUX PROGRAMMING

Processes

www.cce.hcmut.edu.vn

04/10/2022

Le Thanh Van

Outline

- Process Concept
- Process address space
- Process Scheduling
- Process creation
- Process termination
- Signal handling

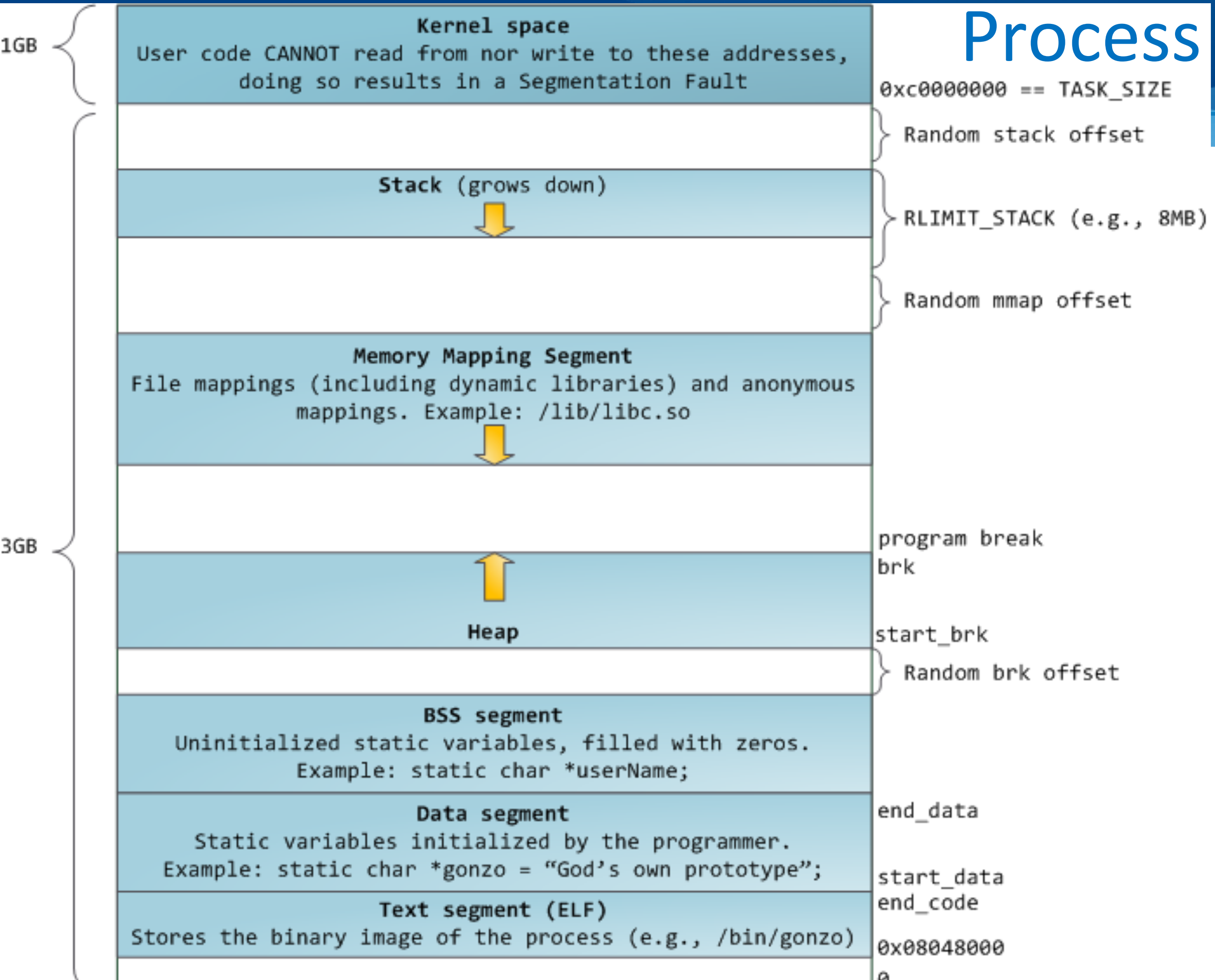
Process Concept

- An operating system executes a variety of programs that run as processes
- ***Process*** – a program in execution; process execution must progress in sequential fashion
- Multiple parts
  - The *program code*, also called ***text** section*
  - Current activity including *program counter*, and *processor registers*
  - ***Stack** section* containing temporary data
    - Function parameters, return addresses, local variables
  - ***Data** section* containing global variables
  - ***Heap** section* containing memory dynamically allocated during run time

Process Concept (Cont.)

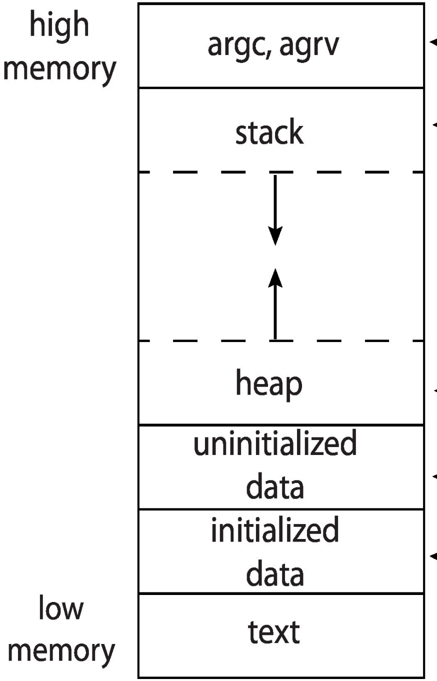
- *Program* is *passive* entity stored on disk (e.g., *executable file*)
- *Process* is *active* entity
  - Program becomes process when executable file loaded into memory
- *Execution of program* can be started via GUI mouse clicks, command line (CLI) entry of its name, etc.
- One program can be several processes
  - E.g., Consider multiple users executing the same program

Process address space



- Text/Code segment contains the binary code of the program
- Data segment contains the initialized global variables and data structures.
- The BSS (Block started by symbol) segment contains the uninitialized global data structures
- Stack segment contains the local variables, return addresses for the particular process. **$ulimit –s** //identify the default stack size

Memory Layout of a C Program




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

int x;
int y = 15;

int main(int argc, char *argv[])
{
    int *values;
    int i;

    values = (int *)malloc(sizeof(int)*5);

    for(i = 0; i < 5; i++)
        values[i] = i;

    return 0;
}
```

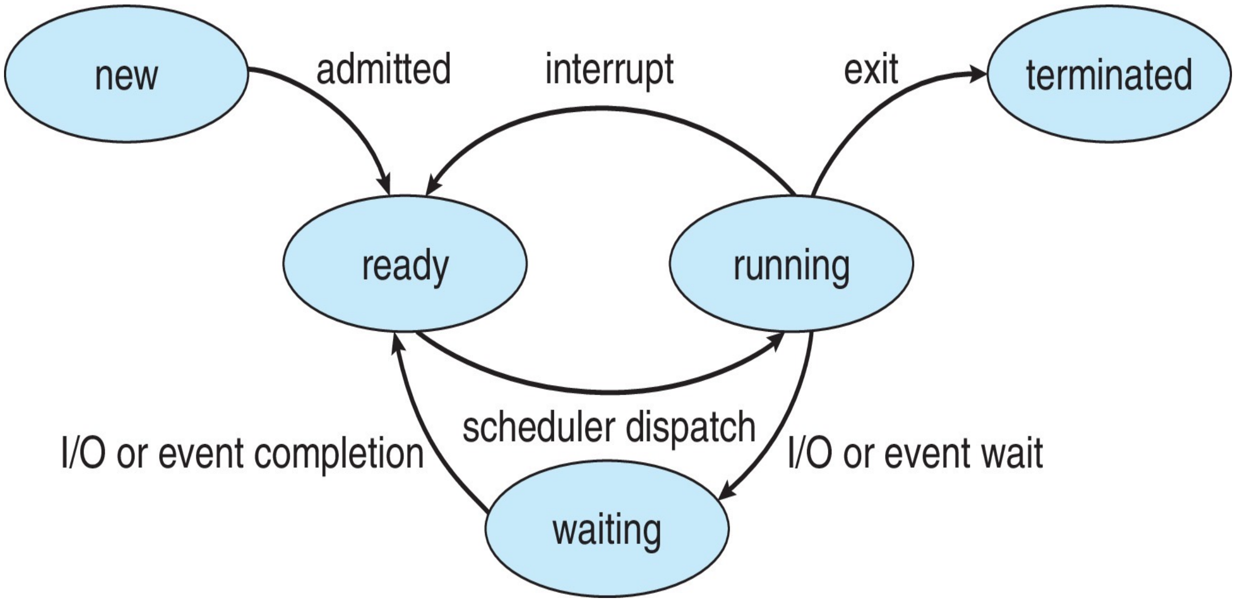
Determine the text/BSS/data/stack segment?

```
#include <iostream>
#include <unistd.h>
int glo_init_data = 9;
int glo_noninit_data;
void print_func (){
    int local_data = 9 ;
    printf( "Process ID = %d\n", getpid());
    printf( "Addresses of the process : \n");
    printf( "  1 . glo_init_data = %p\n", &glo_init_data);
    printf( "  2 . glo_noninit_data = %p\n", &glo_noninit_data);
    printf( "  3 . print_func() = %p\n", &print_func);
    printf( "  4 . local_data = %p\n", &local_data);
}
int main( int argc , char **argv ) {
    print_func();
    return 0;
}
```

Process State

- As a process executes, it changes *state*
  - *New* – The process is being created
  - *Running* – Instructions are being executed
  - *Waiting* – The process is waiting for some event to occur
  - *Ready* – The process is waiting to be assigned to a processor
  - *Terminated* – The process has finished execution

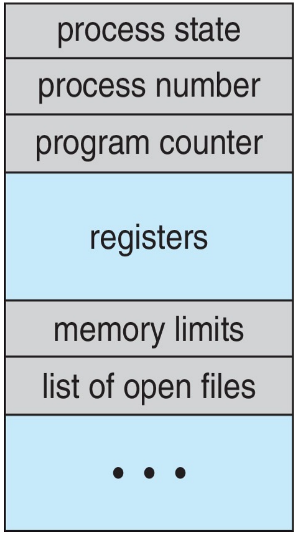
Diagram of Process State



Process Control Block (PCB)

Process Control Block (**PCB**) – Information associated with each process, also called **Task Control Block** (**TCB**), includes:

- *Process state* – running, waiting, etc.
- *Process number* – identity of the process
- *Program counter* – location of instruction to next execute
- *CPU registers* – contents of all process-centric registers
- *CPU scheduling info* – priorities, scheduling queue pointers
- *Memory-management information* – memory allocated to the process
- *Accounting information* – CPU used, clock time elapsed since start, time limits
- *I/O status information* – I/O devices allocated to process, list of open files



Process IDs

- A unique process ID assigned to each new process by OS
- All process has its parent process (except the init process , PID=1)

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

int main ( ){
        printf ("The process ID is %d\n", (int) getpid ()) ;
        printf ("The parent process ID is %d\n", (int) getppid ()) ;
        sleep(100);
        return 0 ;

}
```

```
Vans-MacBook-Air:c-examp ltvan$ ./pid
The process ID is 6790
The parent process ID is 6579
```

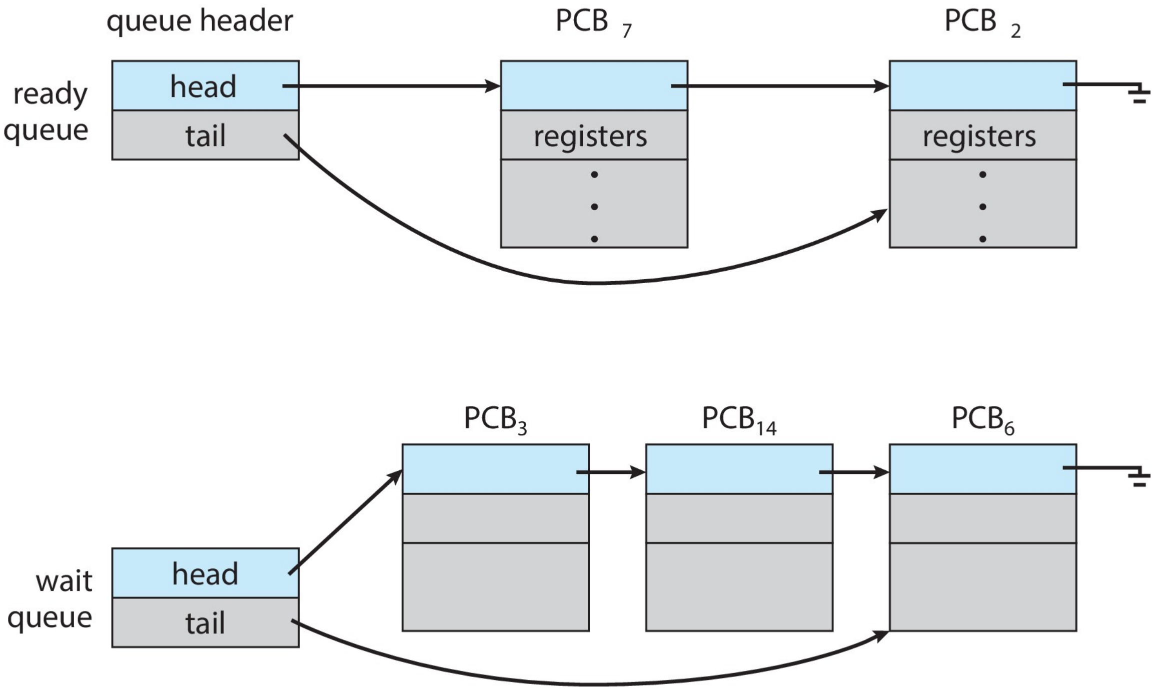
Use htop command to monitor processes and resources of Linux system

```
6579 ltvan     32   0 4196M  1240 ?   0.0  0.0  0:00.00 │      └─ -bash
6714 ltvan     25   0 4168M   696 ?   0.0  0.0  0:00.00 │         └─ ./pid
```

Process Scheduling

- Maximize CPU use ⇢ quickly switch processes onto CPU core
- *Process scheduler* selects one process among available (ready) processes for next execution on CPU core
- Maintains *scheduling queues* of processes
  - *Ready queue* – set of all processes residing in main memory, ready and waiting to execute
  - *Wait queues* – set of processes waiting for an event (e.g., I/O)
- Processes migrate among the various queues

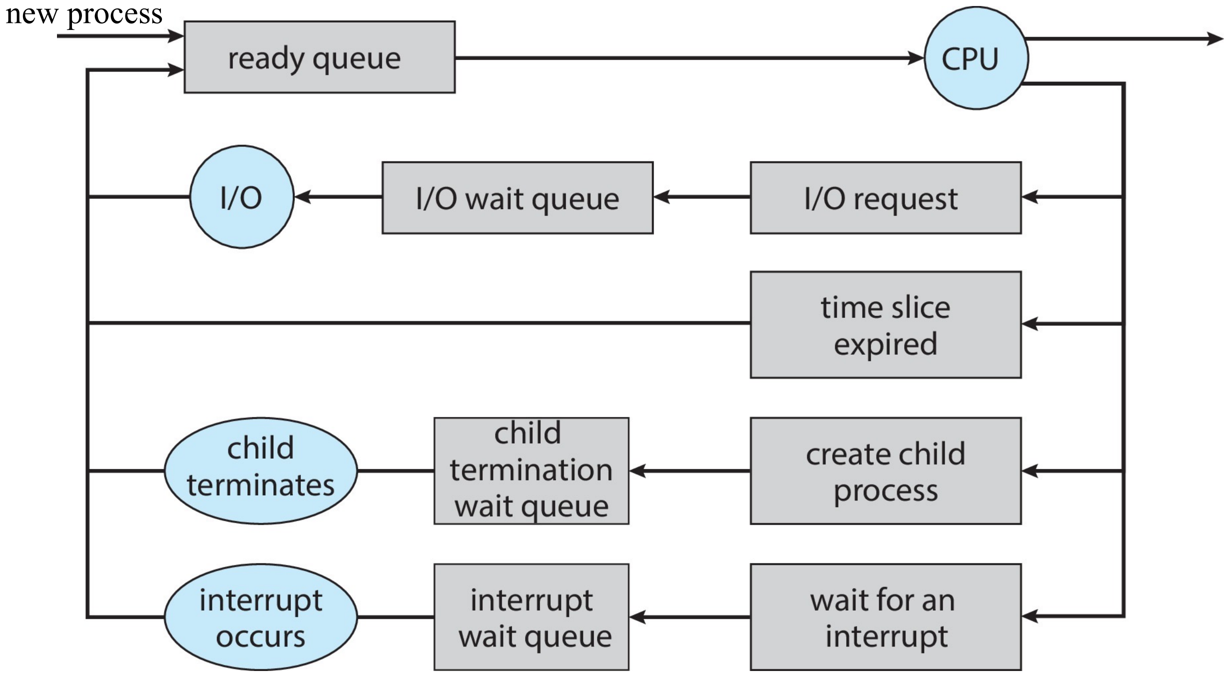
Ready and Wait Queues



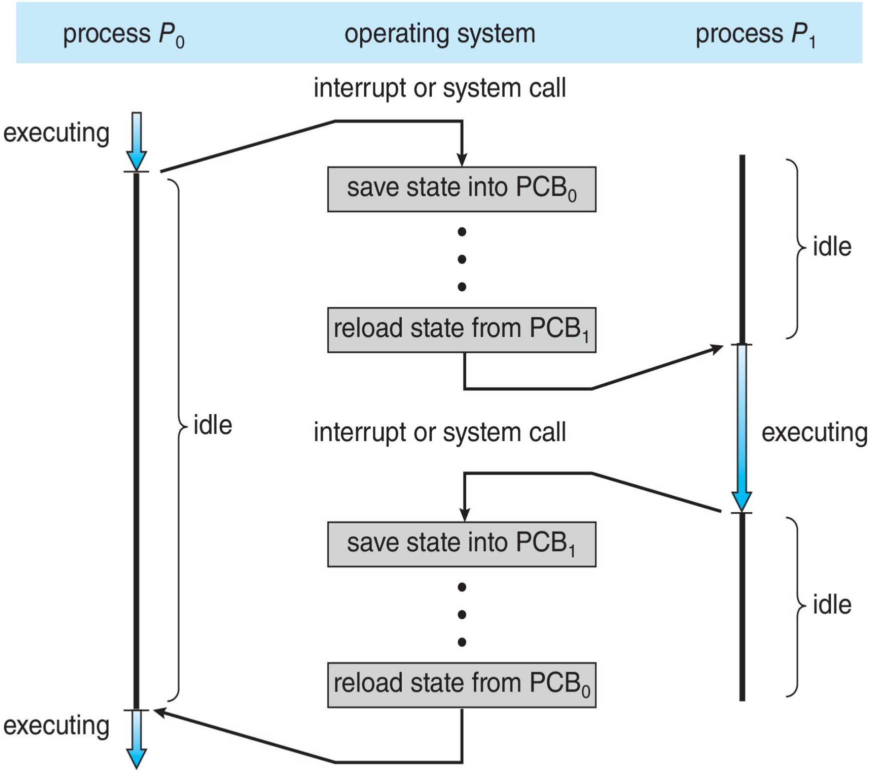
Value “niceness” in process scheduling

- Represent the priority of a process ranged from -20 to +19, 0 by default.
- A higher niceness value means that the process is given a lesser execution priority
- Use **nice command** to assign the niceness of a new process before running:
  - $sudo nice –n <value> [other agruments] <command>
- A positive value of niceness reduces the process’s execution priority.
- Use **renice command** to change the ninceness of a running process from the command line
  - $sudo renice –n <value> [other agruments] -p <PID>

Representation of Process Scheduling



CPU Switch from Process to Process



- A *context switch* occurs when the CPU switches from one process to another.

Context Switch

- When CPU switches to another process, the system must *save the state* of the old process and load the *saved state* for the new process via a *context switch*
- *Context* of a process represented in the **PCB**
- Context-switch time is *overhead,* the system does no useful work while switching
  - The more complex the OS and the PCB, the longer the context switch
- *Time* dependent on hardware support
  - Some hardware provides *multiple sets of registers per CPU*, multiple contexts loaded at once

Operations on Processes

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

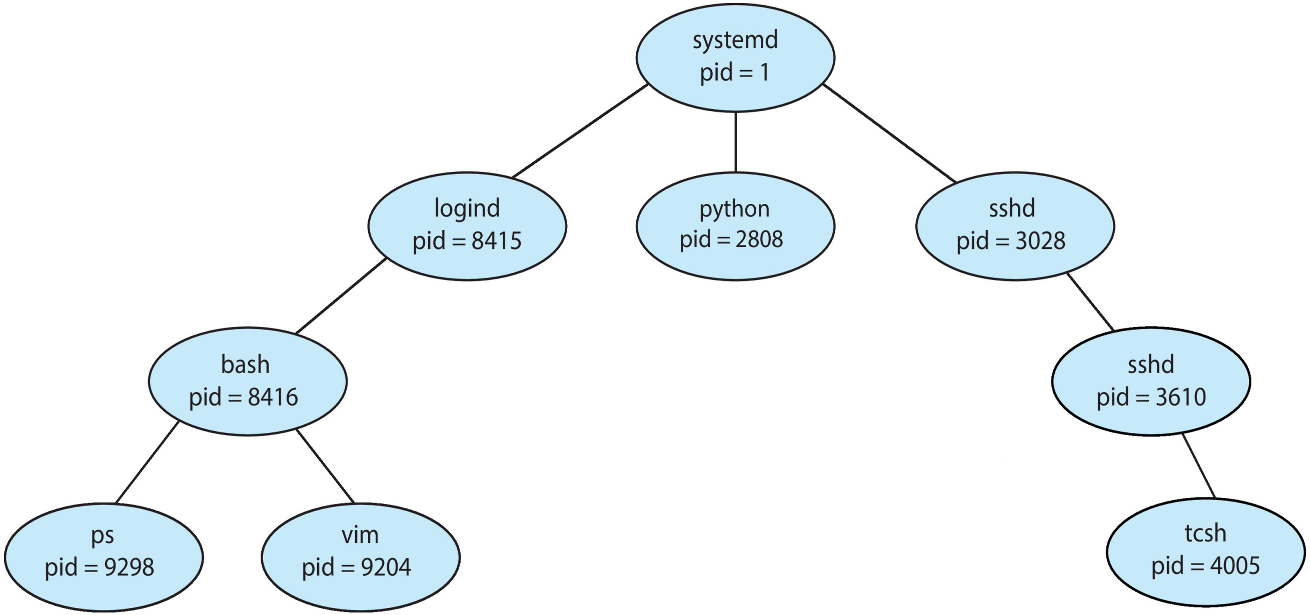
Process Creation

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

Process Creation (Cont.)

- **Execution options**
  - Parent and children execute concurrently
  - Parent waits until children terminate
- **Address space**
  - Child duplicate of parent
  - Child has a program loaded into it

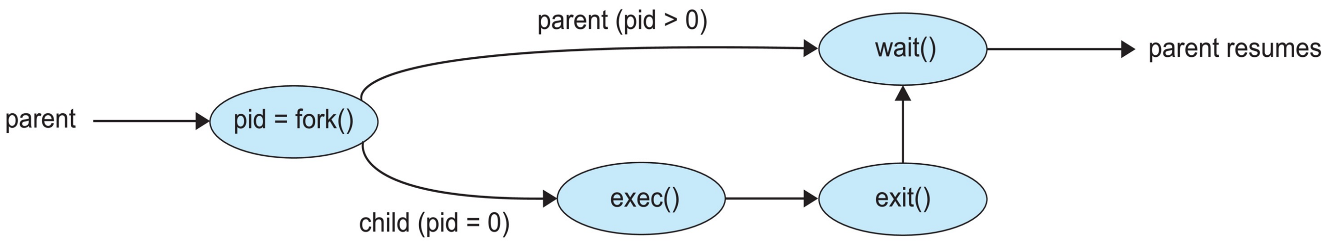
A Tree of Processes in Linux



#pstree

Process Creation (Cont.)

- UNIX examples
  - **`system()`** `to execute a command from within a program`
  - **`fork()`** system call creates new process
  - **`exec()`** system call used after a **`fork()`** to replace the process' memory space with a new program
  - Parent process calls **`wait()`** waiting for the child to terminate



system() system call

```
#include <stdlib.h>
int system(const char *command);
```

- hands the argument command to the command interpreter sh: $/bin/sh –c <command>
  - Ex: $/bin/sh –c ls
- If command is NULL, system() return a nonzero value if a shell is available, or 0 if no shell is available
- If system() return -1: child process could not be created or its status could not be retrieve

```
#include <stdlib.h>
int main ( )
{
int return_value ;
return_value = system ( "ls ." );
return return_value;
}
```

```
Vans-MacBook-Air:c-examp ltvan$ ./systemsimple
2.2.c.rtfd        ex              fork2.c         pid
addrspace         ex.c            forkp           pid.cpp
```

exec() system call

- Replaces the current process image with a new process image.
- Can be used to run a C/C++ prog by using another C/C++ prog.
- exec() family:
- **execl(), execlp(), execle()**
  - `int execl(const char *path, const char *arg, …, NULL);`
  - `int execlp(const char *file, const char *arg, …, NULL );`
- **execv(), execvp(), execvpe()**
  - `int execv(const char *path, char *const argv[]);`
- **execle(), execvpe()**
  - `int execle(const char *path, const char *arg, …, NULL, char * const envp[] );`
- **execlp(), execvp(), execvpe()**

C Program Forking A Separate Process: fork() system call

```
#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;
}
```

Example

- ….
- int main(){
- printf(“Hello \n”);
- 1. fork();
- 2. fork();
- printf(“Hello \n”);
- return 0;
- }

How many processes will be generated?

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 (Cont.)

- 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);`**
  - ✓ **waitpid**() suspends execution of the calling process until a child specified by *pid* argument has changed state
- If no parent waiting (did not invoke **`wait()`**), process is a *zombie*
- If parent terminated without invoking **`wait()`**, process is an *orphan*

wait(&status) example

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
int main (int argc, char *argv[]){
        int     return_code;
        int status;
        /* create a new process */
        return_code = fork();
        if (return_code > 0){


                int id = wait(&status);
                printf("Parent process: child's pid =%d  with returned value = %d \n",id,WEXITSTATUS(status));
                 printf("child's pid =%d  with raw returned value = %d \n",id,status);
                printf("Pid of child %d \n",return_code);
                return 0;
        }
        else if (return_code == 0){
                printf("This is child process \n");
                return 10;//exit(10);
        }
        else {
                printf("Fork error\n");
                return 1;
        }
}
```

#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);

waitpid() example

#include <sys/types.h>
#include <sys/wait.h>

pid_t waitpid(pid_t *pid*, **int** ***status***, **int** *options***);**

- pid< -1: wait for any child process whose process group ID is equal to the absolute value of *pid*.
- pid =1: wait for any child process
- pid = 0: wait for any child process whose process group ID is equal to that of the calling process
- pid>0: wait for the child whose process ID is equal to the value of *pid*

```
#include<stdio.h>
#include<sys/wait.h>
#include<unistd.h>

int main(){
  pid_t pid=fork();
  if(pid==0){
    printf("Child proc id = %d, groupid = %d \n",
getpid(),getpgrp());
  }
  else if(pid>0){
    waitpid(pid,NULL,0);
    printf("Parent proc id = %d \n",getpid());
  }
  return 0;
}
```

wait(&status) is equivalent to waitpid(-1, &status, 0);

```
Vans-MacBook-Air:c-examp ltvan$ ./forkwpidr
Child proc id = 9716, groupid = 9715
Parent proc id = 9715
```

Signals

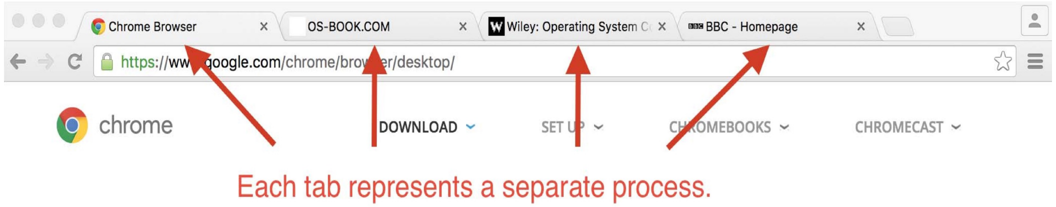
- A special message sent to a process
- Be processed immediately without finishing the current function or even the current line of code.
- Be specified by a signal number or name.
- A default disposition of each signal determines what happens to the process if the program does not specify some other behavior.
- A process may ignore the signal or call a special signal-handler function to respod to the signal.
- SIGBUS, SIGSEGV, SIGFPE: send by Linux system -> terminate the process
- SIGUSR1, SIGUSR2: “user-defined” signal send a command to a running program
- SIGHUP: wake up an idling program or cause a program to reread its config files.
- SIGINT: interrupt from keyboard

Example of signal handling

```
#include<cstdio>
#include<csignal>
#include<unistd.h>
void sig_handler(int signum){
  //Return type of the handler function should be void
  printf("\nInside handler function\n");
}
int main(){
  signal(SIGINT,sig_handler); // Register signal handler
  for(int i=1;i<10;i++){    //Infinite loop
    printf("%d : Inside main function\n",i);
    sleep(1);  // Delay for 1 second
  }
  return 0;
}
```

Multiprocess Architecture – Chrome Browser

- Many web browsers ran as a 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-in



THANK YOU !

Center Of Computer Engineering

www.cce.hcmut.edu.vn