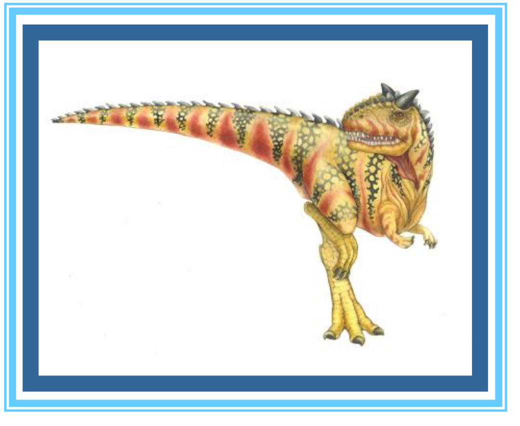
Chapter 3: Processes



Chapter 3: Processes

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- Examples of IPC Systems
- Communication in Client-Server Systems

Objectives

To introduce the notion of a process -- a program in execution, which forms the basis of all computation

To describe the various features of processes, including scheduling, creation and termination, and communication

To explore interprocess communication using shared memory and message passing

To describe communication in client-server systems

Process Concept

- An operating system executes a variety of programs:
  - Batch system – **jobs**
  - Time-shared systems – **user programs** or **tasks**
- Textbook uses the terms ***job*** and ***process*** almost interchangeably
- **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**, processor registers
  - **Stack** containing temporary data
    - Function parameters, return addresses, local variables
  - **Data section** containing global variables
  - **Heap** containing memory dynamically allocated during run time

Process Concept (Cont.)

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

Process in Memory

```
#include "stdafx.h"

int x;
int y = 1;


int _tmain(int argc, _TCHAR* argv[])
{
    int z;
    int *h = new int[100];

    printf("main => %p\n", _tmain);
    printf("x => %p\n", &x);
    printf("y => %p\n", &y);
    printf("z => %p\n", &z);
    printf("h is %p\n", h);

    getchar();

    return 0;
}
```

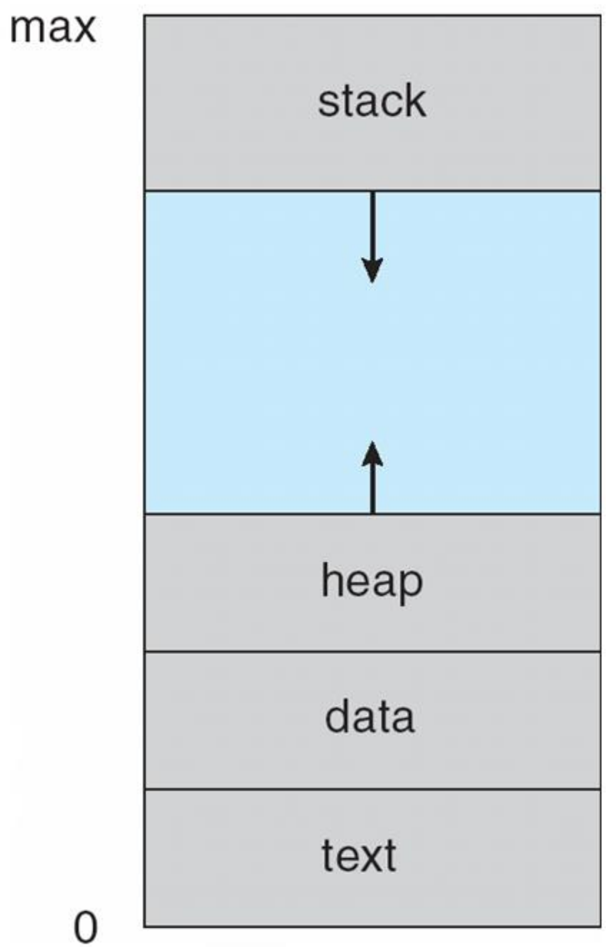
Process in Memory

```
     3:
     4: #include "stdafx.h"
     5:
     6: int x;
     7: int y = 1;
     8:
     9:
    10: int _tmain(int argc, _TCHAR* argv[])
    11: {
000007F7FB402E50 48 89 54 24 10       mov         qword ptr [rsp+10h],rdx
000007F7FB402E55 89 4C 24 08          mov         dword ptr [rsp+8],ecx
000007F7FB402E59 57                   push        rdi
000007F7FB402E5A 48 83 EC 50          sub         rsp,50h
000007F7FB402E5E 48 8B FC             mov         rdi,rsp
000007F7FB402E61 B9 14 00 00 00       mov         ecx,14h
000007F7FB402E66 B8 CC CC CC CC       mov         eax,0CCCCCCCCh
000007F7FB402E6B F3 AB                rep stos    dword ptr [rdi]
000007F7FB402E6D 8B 4C 24 60          mov         ecx,dword ptr [rsp+60h]
    12:     int z;
    13:     int *h = new int[100];
000007F7FB402E71 B9 90 01 00 00       mov         ecx,190h
000007F7FB402E76 E8 DB E2 FF FF       call        operator new (7F7FB401156h)
000007F7FB402E7B 48 89 44 24 40       mov         qword ptr [rsp+40h],rax
000007F7FB402E80 48 8B 44 24 40       mov         rax,qword ptr [rsp+40h]
000007F7FB402E85 48 89 44 24 38       mov         qword ptr [h],rax
    14:
    15:     printf("main => %p\n", _tmain);
000007F7FB402E8A 48 8D 15 74 E1 FF FF lea         rdx,[@ILT+0(wmain) (7F7FB401005h)]
000007F7FB402E91 48 8D 0D F8 38 00 00 lea         rcx,[__xi_z+130h (7F7FB406790h)]
000007F7FB402E98 FF 15 72 86 00 00    call        qword ptr [__imp_printf (7F7FB40B510h)]
    16:     printf("x => %p\n", &x);
000007F7FB402E9E 48 8D 15 AB 62 00 00 lea         rdx,[x (7F7FB409150h)]
000007F7FB402EA5 48 8D 0D F4 38 00 00 lea         rcx,[__xi_z+140h (7F7FB4067A0h)]
000007F7FB402EAC FF 15 5E 86 00 00    call        qword ptr [__imp_printf (7F7FB40B510h)]
    17:     printf("y => %p\n", &y);
000007F7FB402EB2 48 8D 15 47 61 00 00 lea         rdx,[y (7F7FB409000h)]
000007F7FB402EB9 48 8D 0D F0 38 00 00 lea         rcx,[__xi_z+150h (7F7FB4067B0h)]
000007F7FB402EC0 FF 15 4A 86 00 00    call        qword ptr [__imp_printf (7F7FB40B510h)]
```

Process in Memory

```
    18:     printf("z => %p\n", &z);
000007F7FB402EC6 48 8D 54 24 24       lea         rdx,[z]
000007F7FB402ECB 48 8D 0D EE 38 00 00 lea         rcx,[__xi_z+160h (7F7FB4067C0h)]
000007F7FB402ED2 FF 15 38 86 00 00    call        qword ptr [__imp_printf (7F7FB40B510h)]
    19:     printf("h is %p\n", h);
000007F7FB402ED8 48 8B 54 24 38       mov         rdx,qword ptr [h]
000007F7FB402EDD 48 8D 0D EC 38 00 00 lea         rcx,[__xi_z+170h (7F7FB4067D0h)]
000007F7FB402EE4 FF 15 26 86 00 00    call        qword ptr [__imp_printf (7F7FB40B510h)]
    20:
    21:     getchar();
000007F7FB402EEA FF 15 28 86 00 00    call        qword ptr [__imp_getchar (7F7FB40B518h)]
    22:
    23:     return 0;
000007F7FB402EF0 33 C0                xor         eax,eax
    24: }
000007F7FB402EF2 8B F8                mov         edi,eax
000007F7FB402EF4 48 8B CC             mov         rcx,rsp
000007F7FB402EF7 48 8D 15 22 39 00 00 lea         rdx,[__xi_z+1C0h (7F7FB406820h)]
000007F7FB402EFE E8 CD E1 FF FF       call        _RTC_CheckStackVars (7F7FB4010D0h)
000007F7FB402F03 8B C7                mov         eax,edi
000007F7FB402F05 48 83 C4 50          add         rsp,50h
000007F7FB402F09 5F                   pop         rdi
000007F7FB402F0A C3                   ret
```

Process in Memory

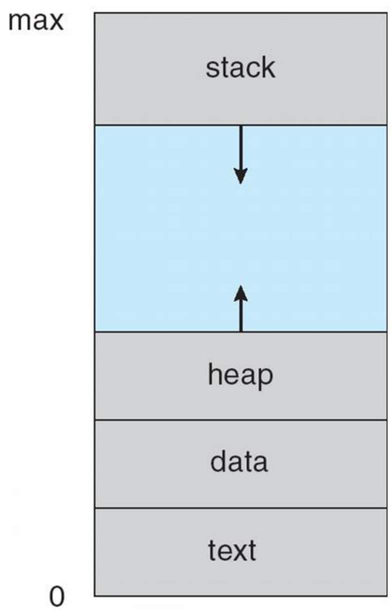


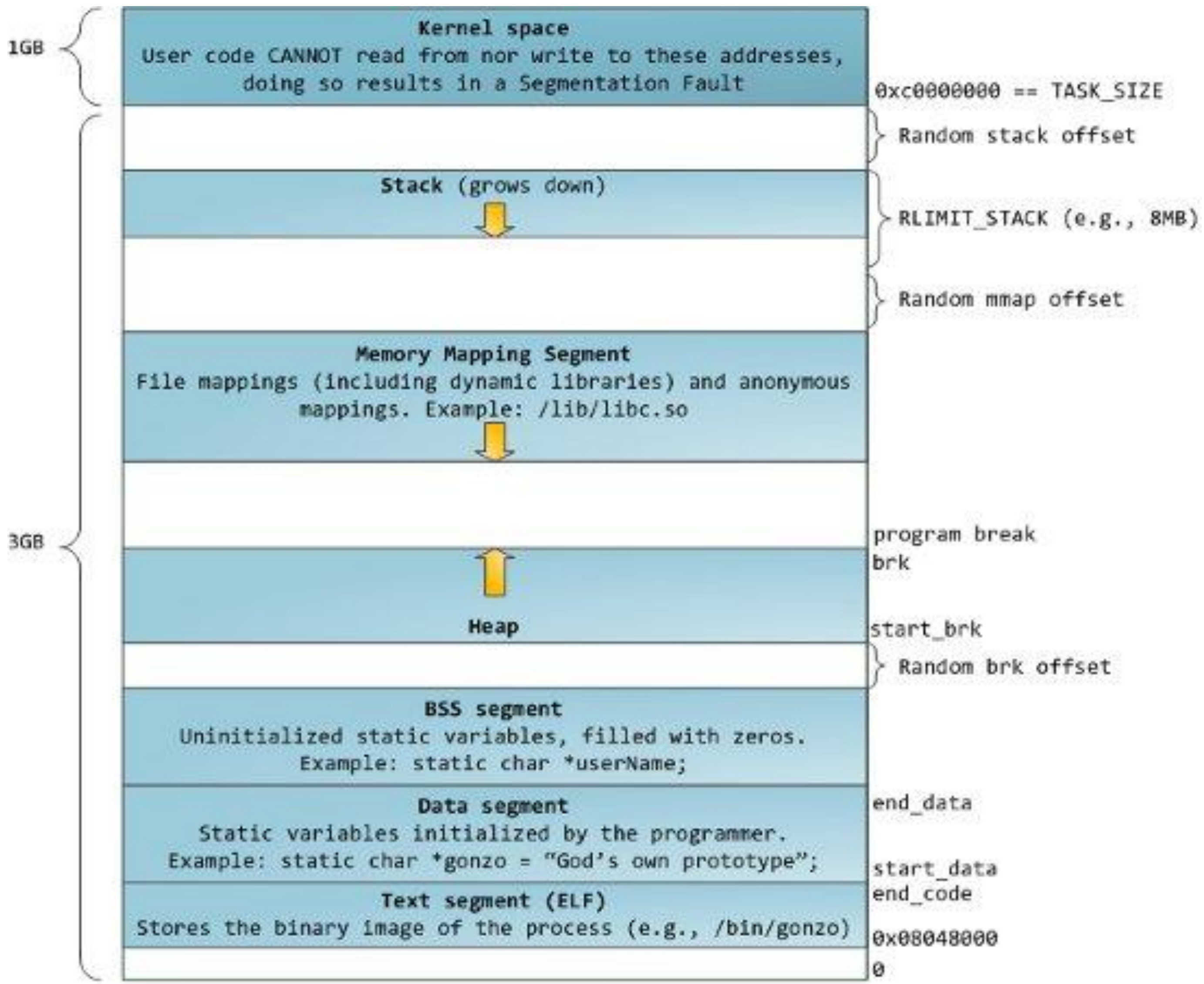
c:\users\hank\documents\visual studio 2010\Projects\test_process1\x64\...

```
main => 000007F7FB401005
x => 000007F7FB409150
y => 000007F7FB409000
z => 000000000097FAD4
h is 0000000000AD83A0
```

Address	Type	Size	Protection	Details
+ 0000000000850000	Heap (Shareable)	64 K	Read/Write	Heap ID: 2 [COMPATABILITY]
+ 0000000000860000	Mapped File	4 K	Read	C:\Windows\Globalization\zh-TW.nlx
+ 0000000000870000	Shareable	36 K	Read	
− 0000000000880000	Thread Stack	1,024 K	Read/Write/Guard	Thread ID: 1340
0000000000880000	Thread Stack	992 K	Reserved	
0000000000978000	Thread Stack	12 K	Read/Write/Guard	
000000000097B000	Thread Stack	20 K	Read/Write	
+ 0000000000980000	Shareable	16 K	Read	
+ 0000000000990000	Shareable	4 K	Read	
+ 00000000009A0000	Private Data	8 K	Read/Write	
+ 00000000009B0000	Mapped File	468 K	Read	C:\Windows\System32\locale.nls
− 0000000000AD0000	Heap (Private Data)	64 K	Read/Write	Heap ID: 3 [COMPATABILITY]
0000000000AD0000	Heap (Private Data)	36 K	Read/Write	Heap ID: 3 [COMPATABILITY]
0000000000AD9000	Heap (Private Data)	28 K	Reserved	Heap ID: 3 [COMPATABILITY]
+ 0000000000B50000	Heap (Private Data)	1,024 K	Read/Write	Heap ID: 1 [COMPATABILITY]
+ 000000005B3E0000	Image (ASLR)	1,856 K	Execute/Read	C:\Windows\System32\msvcr100d.dll
+ 000000007FFE0000	Private Data	64 K	Read	
+ 000007F7FB0F0000	Shareable	1,024 K	Read	
+ 000007F7FB1F0000	Shareable	204 K	Read	
+ 000007F7FB223000	Private Data	4 K	Read/Write	Process Environment Block
+ 000007F7FB22E000	Private Data	8 K	Read/Write	Thread Environment Block ID: 1340
− 000007F7FB400000	Image (ASLR)	56 K	Execute/Read	C:\Users\Hank\Documents\Visual Studio 2010\Projects\test_pr
000007F7FB400000	Image (ASLR)	4 K	Read	Header
000007F7FB401000	Image (ASLR)	20 K	Execute/Read	.text
000007F7FB406000	Image (ASLR)	12 K	Read	.rdata
000007F7FB409000	Image (ASLR)	4 K	Read/Write	.data
000007F7FB40A000	Image (ASLR)	4 K	Read	.pdata
000007F7FB40B000	Image (ASLR)	4 K	Read/Write	.idata
000007F7FB40C000	Image (ASLR)	4 K	Read	.rsrc
000007F7FB40D000	Image (ASLR)	4 K	Read	.reloc
+ 000007FDCCDA0000	Image (ASLR)	972 K	Execute/Read	C:\Windows\System32\KernelBase.dll

Process in Memory





Process State

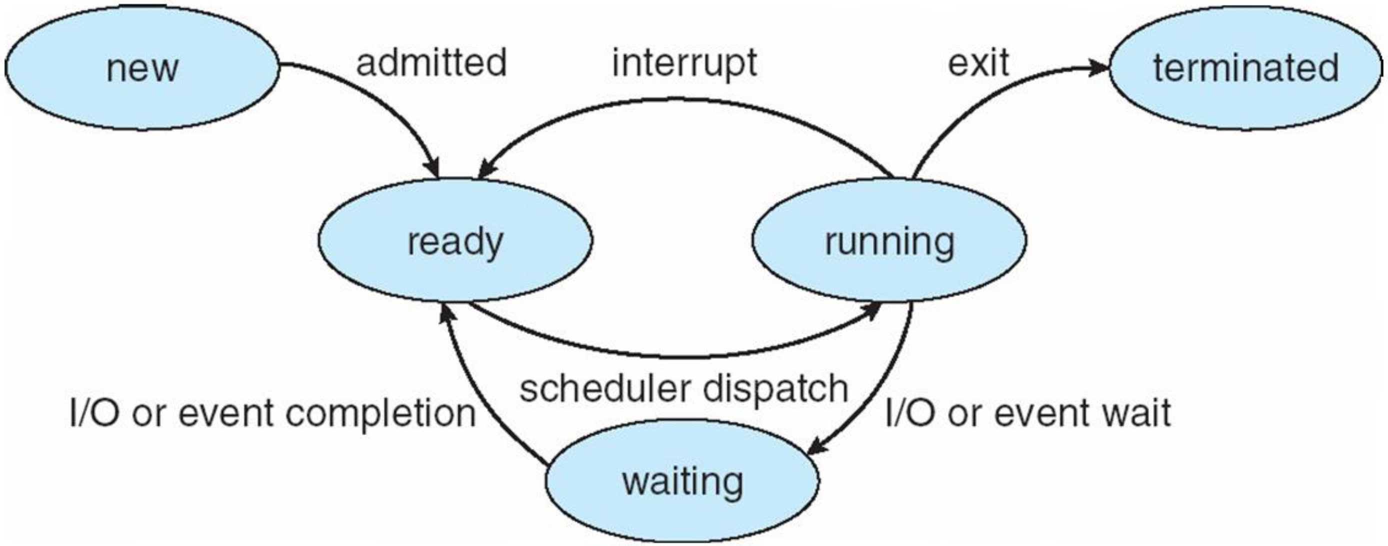
As a process executes, it changes *state*

- **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

```
hank@Maestro:/home/hank
File  Edit  View  Search  Terminal  Help
top - 23:19:48 up 3 min,  2 users,  load average: 0.56, 0.52, 0.25
Tasks: 144 total,   2 running, 142 sleeping,   0 stopped,   0 zombie
Cpu(s): 10.3%us,  7.4%sy,  0.0%ni, 81.9%id,  0.2%wa,  0.0%hi,  0.2%si,  0.0%st
Mem:   1019704k total,   878904k used,   140800k free,    34432k buffers
Swap:  2064380k total,        0k used,  2064380k free,   318172k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 2669 hank      20   0  721m 100m  31m S 19.3 10.0   0:15.16 firefox
 1936 root      20   0  303m  93m  14m S 12.0  9.4   0:07.24 Xorg
 2268 hank      20   0 1836m  94m  43m S 10.6  9.5   0:08.35 gnome-shell
 2477 hank      20   0  572m  15m 9884 S  0.3  1.5   0:00.55 gnome-terminal
 2668 root      20   0 15256 1212  900 R  0.3  0.1   0:00.21 top
    1 root      20   0 66744  24m 2080 S  0.0  2.5   0:01.20 systemd
    2 root      20   0     0    0    0 S  0.0  0.0   0:00.00 kthreadd
    3 root      20   0     0    0    0 S  0.0  0.0   0:00.01 ksoftirqd/0
    4 root      20   0     0    0    0 S  0.0  0.0   0:00.00 kworker/0:0
    5 root      20   0     0    0    0 S  0.0  0.0   0:00.00 kworker/u:0
    6 root      RT   0     0    0    0 S  0.0  0.0   0:00.00 migration/0
    7 root      RT   0     0    0    0 S  0.0  0.0   0:00.00 watchdog/0
    8 root      RT   0     0    0    0 S  0.0  0.0   0:00.00 migration/1
    9 root      20   0     0    0    0 S  0.0  0.0   0:00.00 kworker/1:0
   10 root      20   0     0    0    0 S  0.0  0.0   0:00.04 ksoftirqd/1
   11 root      RT   0     0    0    0 S  0.0  0.0   0:00.00 watchdog/1
   12 root       0 -20     0    0    0 S  0.0  0.0   0:00.00 cpuset
   13 root       0 -20     0    0    0 S  0.0  0.0   0:00.00 khelper
```

Diagram of Process State



Process State

hank@Maestro:/home/ha

File Edit View Search Terminal Help

```
[root@Maestro hank]# ps axfj | more
 PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
    0     2     0     0 ?           -1 S        0   0:00 [kthreadd]
    2     3     0     0 ?           -1 S        0   0:00  \_ [ksoftirqd/0]
    2     5     0     0 ?           -1 S        0   0:00  \_ [kworker/u:0]
    2     6     0     0 ?           -1 S        0   0:00  \_ [migration/0]
    2     7     0     0 ?           -1 S        0   0:00  \_ [watchdog/0]
    2     8     0     0 ?           -1 S        0   0:00  \_ [migration/1]
    2    10     0     0 ?           -1 S        0   0:00  \_ [ksoftirqd/1]
    2    11     0     0 ?           -1 S        0   0:00  \_ [watchdog/1]
    2    12     0     0 ?           -1 S<       0   0:00  \_ [cpuset]
    2    13     0     0 ?           -1 S<       0   0:00  \_ [khelper]
    2    14     0     0 ?           -1 S        0   0:00  \_ [kdevtmpfs]
    2    15     0     0 ?           -1 S<       0   0:00  \_ [netns]
    2    16     0     0 ?           -1 S        0   0:00  \_ [sync_supers]
    2    17     0     0 ?           -1 S        0   0:00  \_ [bdi-default]
    2    18     0     0 ?           -1 S<       0   0:00  \_ [kintegrityd]
    2    19     0     0 ?           -1 S<       0   0:00  \_ [kblockd]
    2    20     0     0 ?           -1 S<       0   0:00  \_ [ata_sff]
    2    21     0     0 ?           -1 S        0   0:00  \_ [khubd]
    2    22     0     0 ?           -1 S<       0   0:00  \_ [md]
    2    23     0     0 ?           -1 S        0   0:00  \_ [kworker/1:1]
    2    25     0     0 ?           -1 S        0   0:00  \_ [kswapd0]
    2    26     0     0 ?           -1 SN       0   0:00  \_ [ksmd]
    2    27     0     0 ?           -1 SN       0   0:00  \_ [khugepaged]
    2    28     0     0 ?           -1 S        0   0:00  \_ [fsnotify_mark]
    2    29     0     0 ?           -1 S<       0   0:00  \_ [crypto]
    2    35     0     0 ?           -1 S<       0   0:00  \_ [kthrotld]
    2    38     0     0 ?           -1 S        0   0:00  \_ [scsi_eh_0]
    2    39     0     0 ?           -1 S        0   0:00  \_ [scsi_eh_1]
    2    40     0     0 ?           -1 S        0   0:00  \_ [scsi_eh_2]
    2    41     0     0 ?           -1 S        0   0:00  \_ [kworker/u:2]
    2    43     0     0 ?           -1 S<       0   0:00  \_ [kpsmoused]
    2    44     0     0 ?           -1 S<       0   0:00  \_ [deferwq]
    2    46     0     0 ?           -1 S        0   0:00  \_ [kworker/0:2]
    2   238     0     0 ?           -1 S        0   0:00  \_ [kworker/1:2]
    2   290     0     0 ?           -1 S<       0   0:00  \_ [kdmflush]
    2   291     0     0 ?           -1 S<       0   0:00  \_ [kdmflush]
    2   338     0     0 ?           -1 S        0   0:00  \_ [jbd2/dm-1-8]
    2   339     0     0 ?           -1 S<       0   0:00  \_ [ext4-dio-unwrit]
    2   375     0     0 ?           -1 S        0   0:00  \_ [kauditd]
```

UNIX-LIKE FOR LIFE

Friday, July 31, 2009

Linux process state codes

Here are the different values that the s, stat and state output specifiers (header "STAT" or "S") will display to describe the state of a process.

D Uninterruptible sleep (usually IO)

R Running or runnable (on run queue)

S Interruptible sleep (waiting for an event to complete)

T Stopped, either by a job control signal or because it is being traced.

W paging (not valid since the 2.6.xx kernel)

X dead (should never be seen)

Z Defunct ("zombie") process, terminated but not reaped by its parent.

For BSD formats and when the stat keyword is used, additional characters may be displayed:

< high-priority (not nice to other users)

N low-priority (nice to other users)

L has pages locked into memory (for real-time and custom IO)

s is a session leader

l is multi-threaded (using CLONE_THREAD, like NPTL pthreads do)

+ is in the foreground process group

Posted by M.Burak Alkan at 12:01 AM

Labels linux

No comments:

Post a Comment

Newer Post Home Old

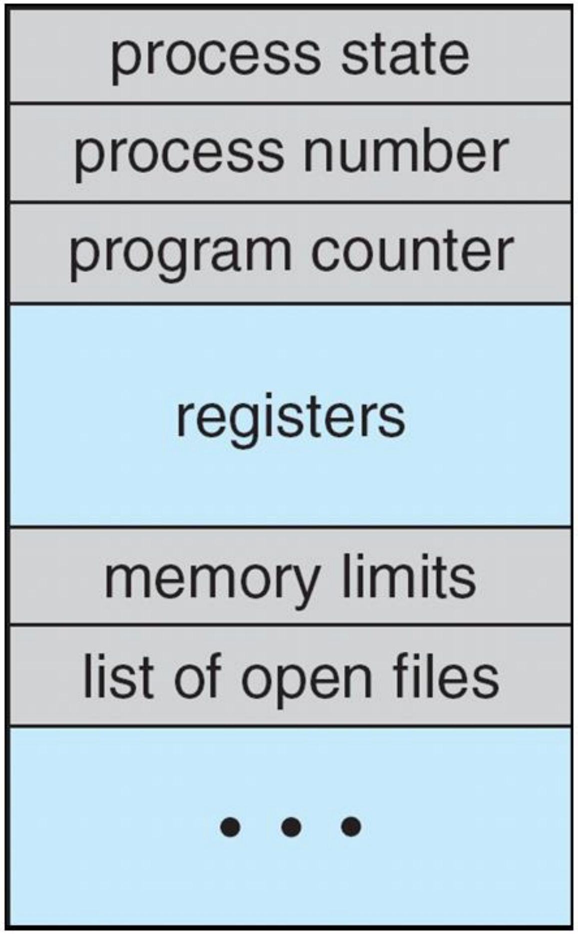
Subscribe to: Post Comments (Atom)

Process Control Block (PCB)

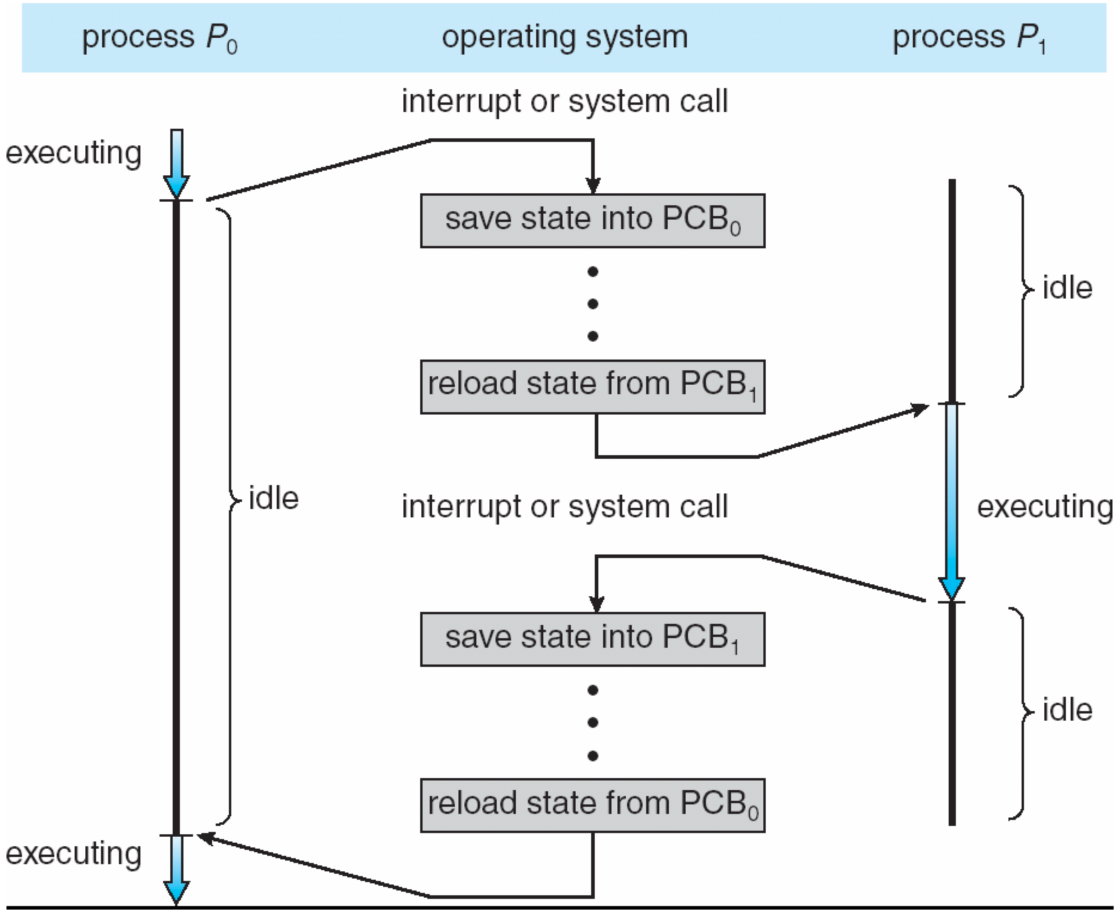
Information associated with each process

(also called **task control block**)

- Process state – running, waiting, etc
- Program counter – location of instruction to next execute
- CPU registers – contents of all process-centric registers
- CPU scheduling information- 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

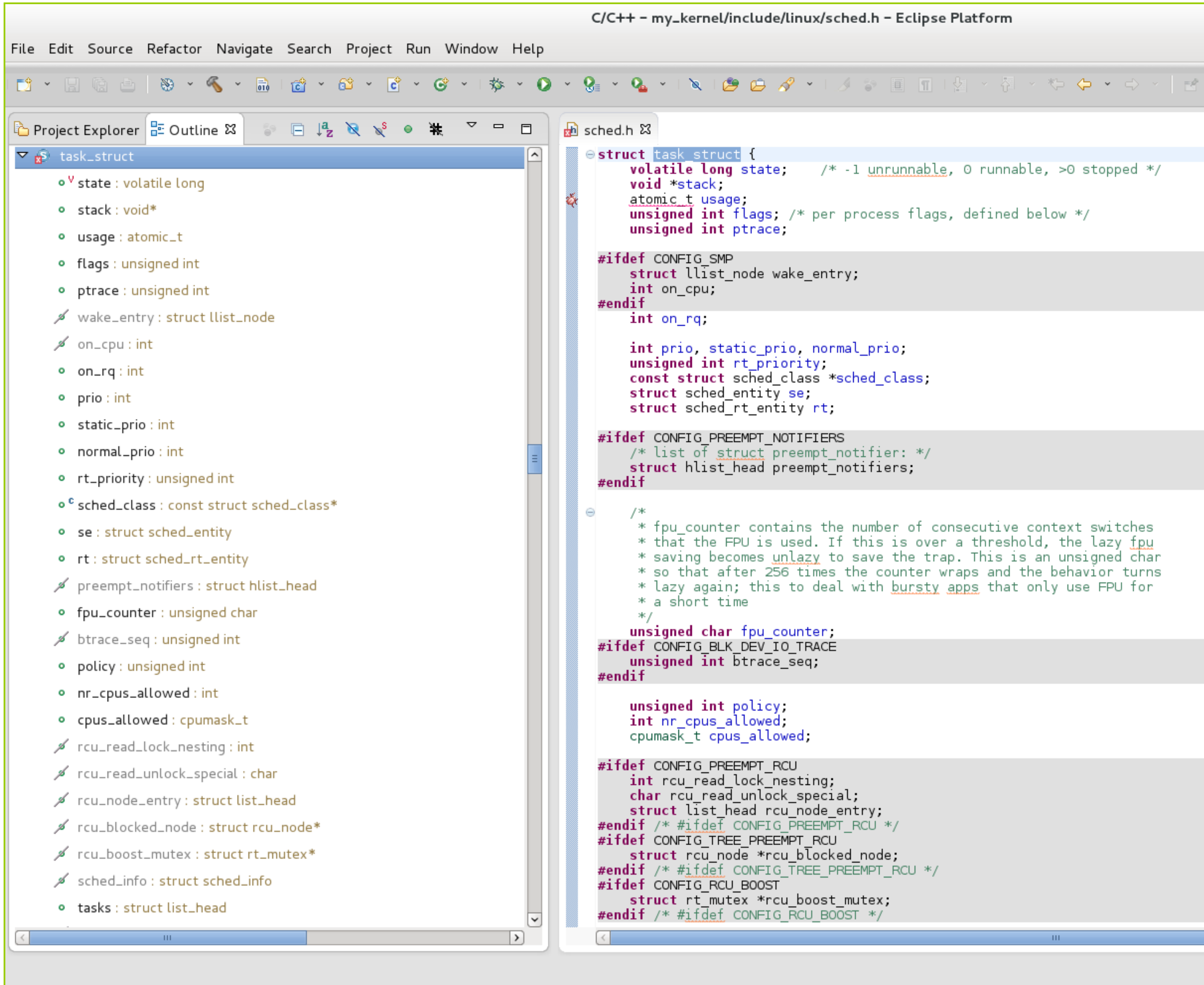


CPU Switch From Process to Process



Process Control Block (PCB)

include/linux/sched.h

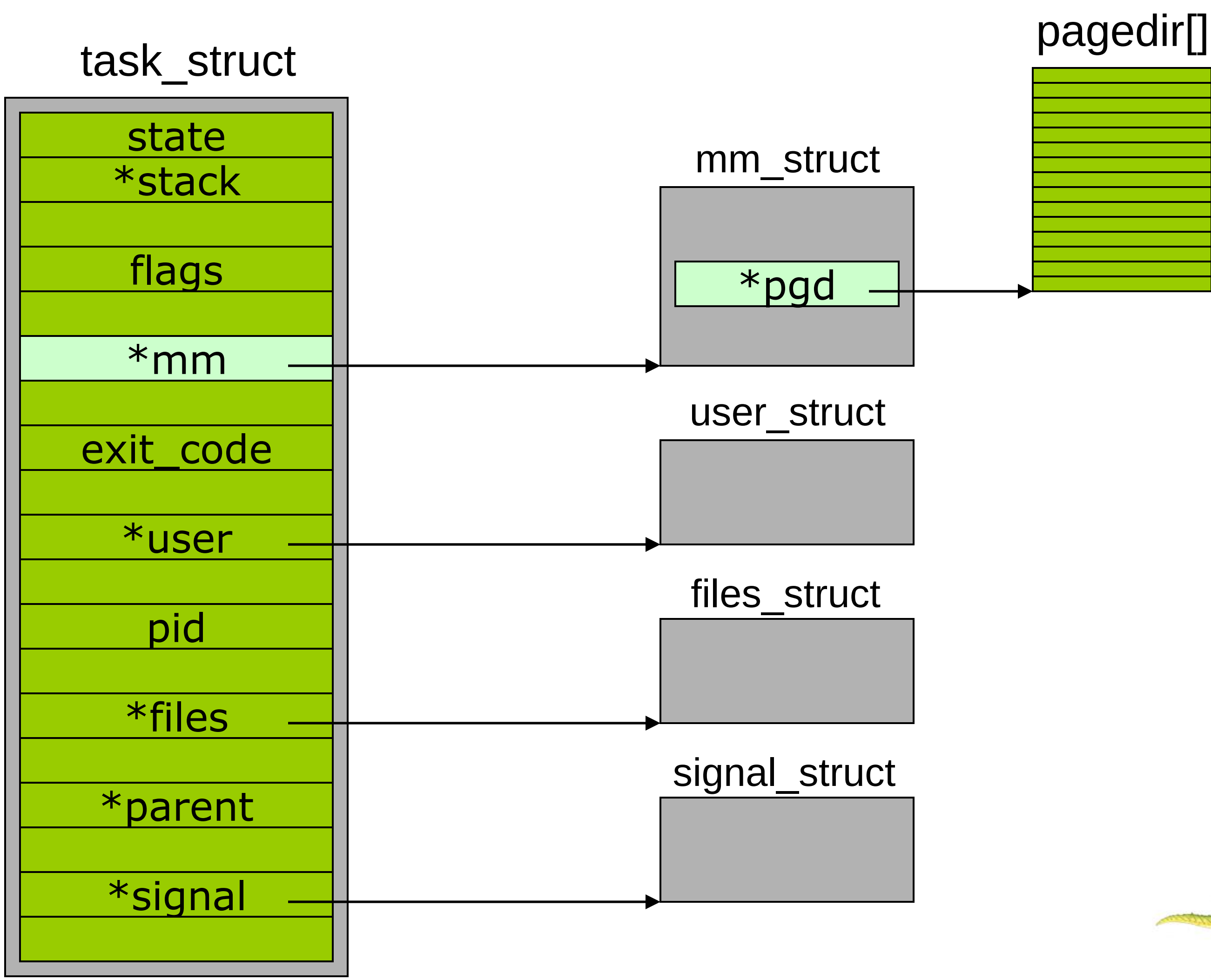


The Linux process descriptor

Each process descriptor contains many fields

and some are pointers to other kernel structures

which may themselves include fields that point to structures



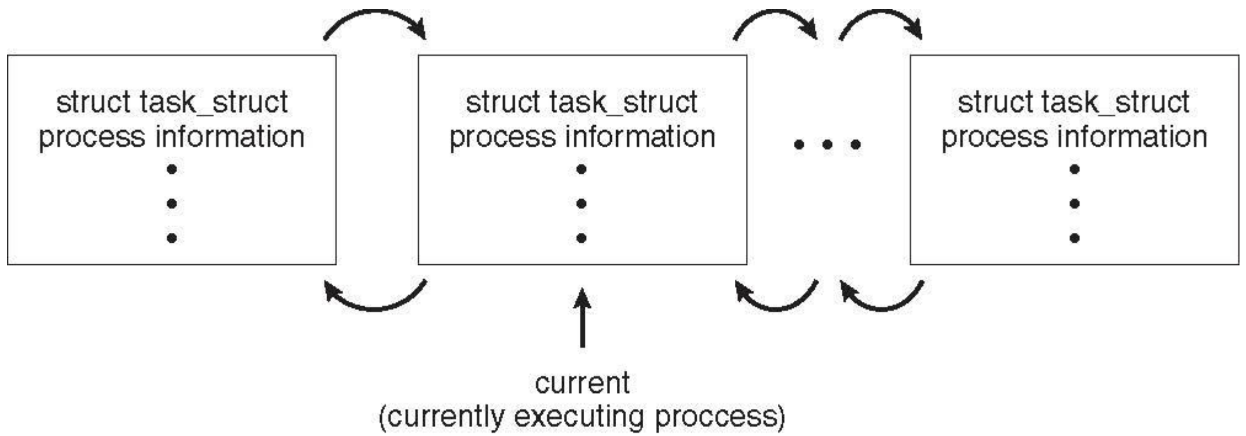
Threads

- So far, process has a single thread of execution
- Consider having multiple program counters per process
  - Multiple locations can execute at once
    - Multiple threads of control -> **threads**
- Must then have storage for thread details, multiple program counters in PCB
- See next chapter

Process Representation in Linux

Represented by the C structure `task_struct`

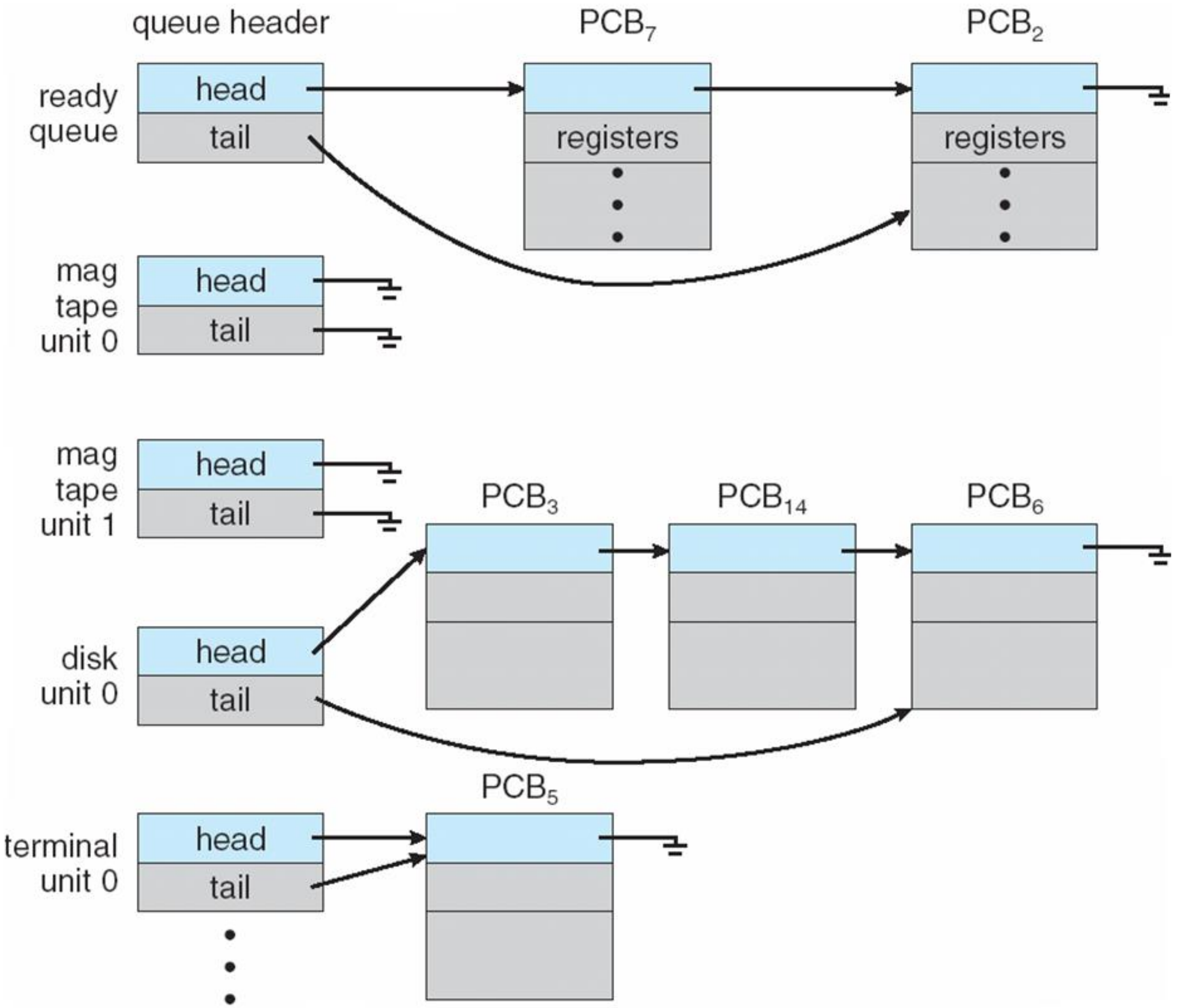
```
pid t_pid; /* process identifier */
long state; /* state of the process */
unsigned int time_slice /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files; /* list of open files */
struct mm_struct *mm; /* address space of this process */
```



Process Scheduling

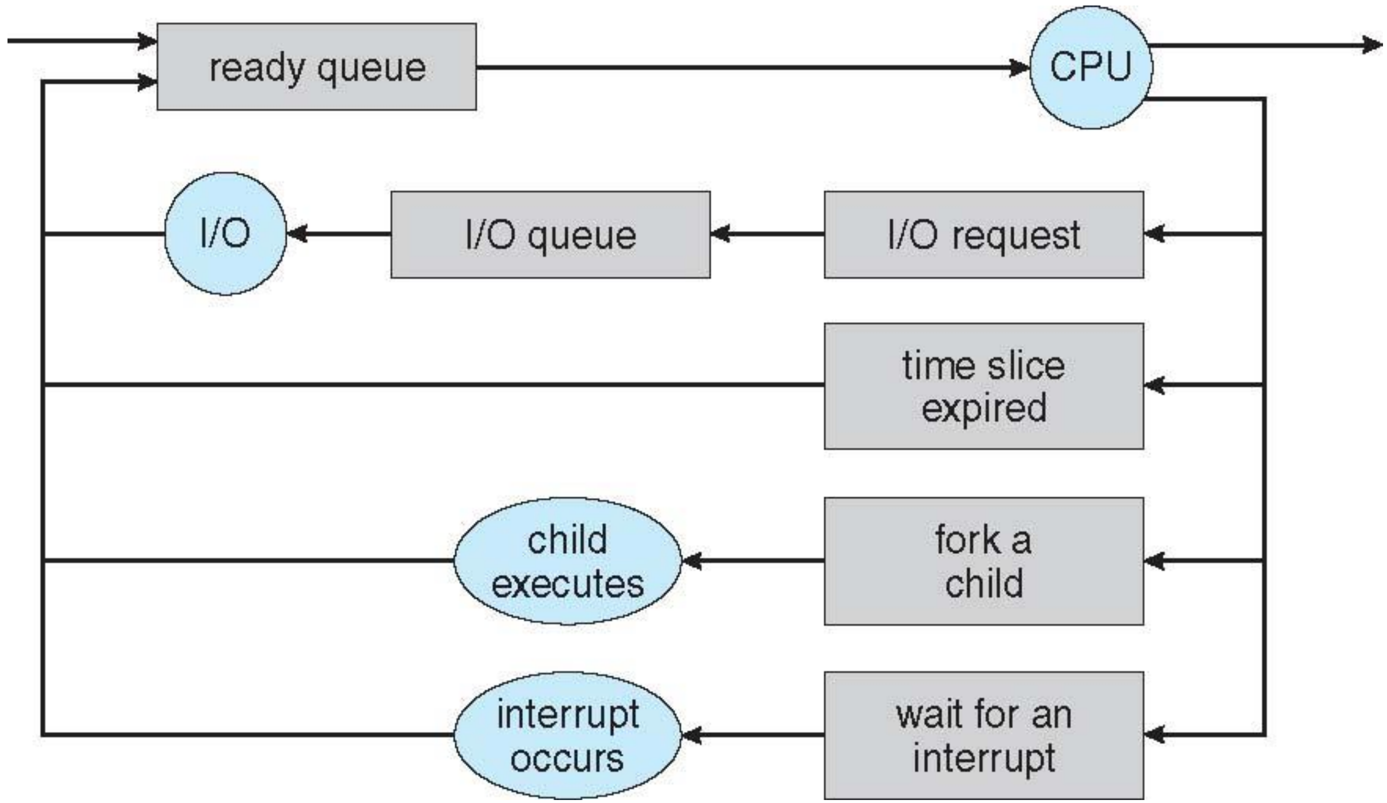
- Maximize CPU use, quickly switch processes onto CPU for time sharing
- **Process scheduler** selects among available processes for next execution on CPU
- Maintains **scheduling queues** of processes
  - **Job queue** – set of all processes in the system
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - **Device queues** – set of processes waiting for an I/O device
  - Processes migrate among the various queues

Ready Queue And Various I/O Device Queues



Representation of Process Scheduling

Queueing diagram represents queues, resources, flows



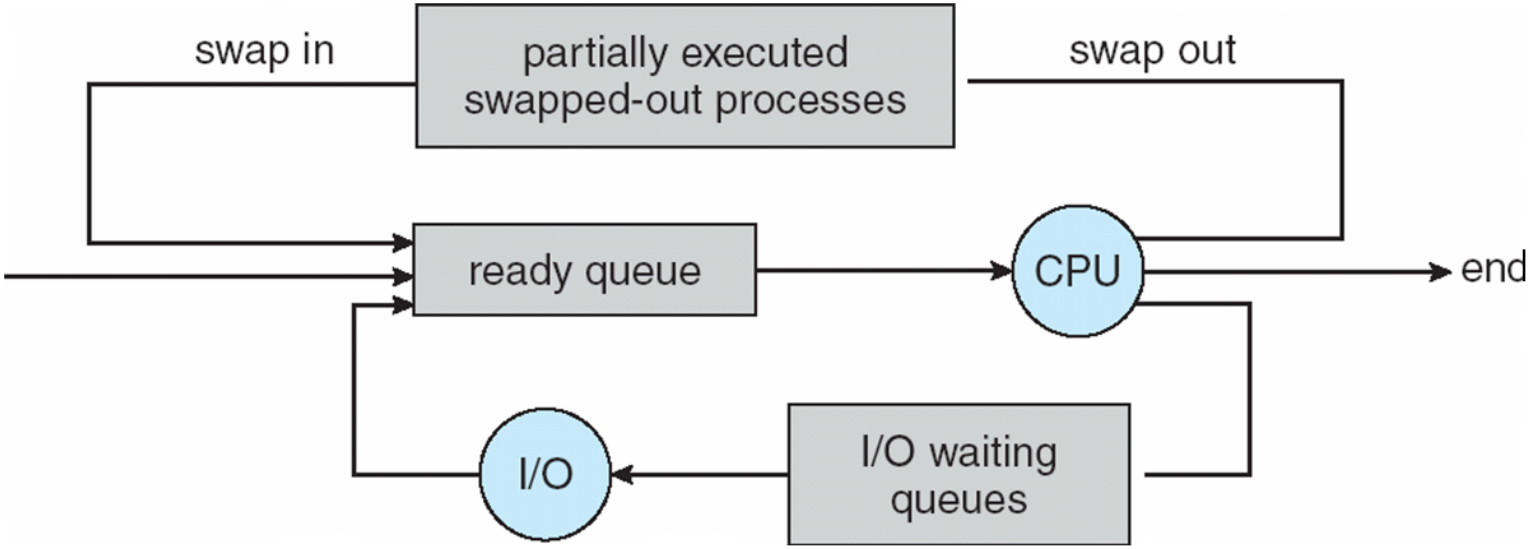
Schedulers

- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
  - Short-term scheduler is invoked frequently (milliseconds) $\Rightarrow$ (must be fast)
- **Long-term scheduler** (or **job scheduler**) – selects which processes should be brought into the ready queue
  - Long-term scheduler is invoked infrequently (seconds, minutes) $\Rightarrow$ (may be slow)
  - The long-term scheduler controls the **degree of multiprogramming**
- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts
- Long-term scheduler strives for good ***process mix***

Addition of Medium Term Scheduling

Medium-term scheduler can be added if degree of multiple programming needs to decrease

- Remove process from memory, store on disk, bring back in from disk to continue execution: **swapping**



Multitasking in Mobile Systems

- Some mobile systems (e.g., early version of iOS) allow only one process to run, others suspended
- Due to screen real estate, user interface limits iOS provides for a
  - Single **foreground** process- controlled via user interface
  - Multiple **background** processes– in memory, running, but not on the display, and with limits
  - Limits include single, short task, receiving notification of events, specific long-running tasks like audio playback
- Android runs foreground and background, with fewer limits
  - Background process uses a **service** to perform tasks
  - Service can keep running even if background process is suspended
  - Service has no user interface, small memory use

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,

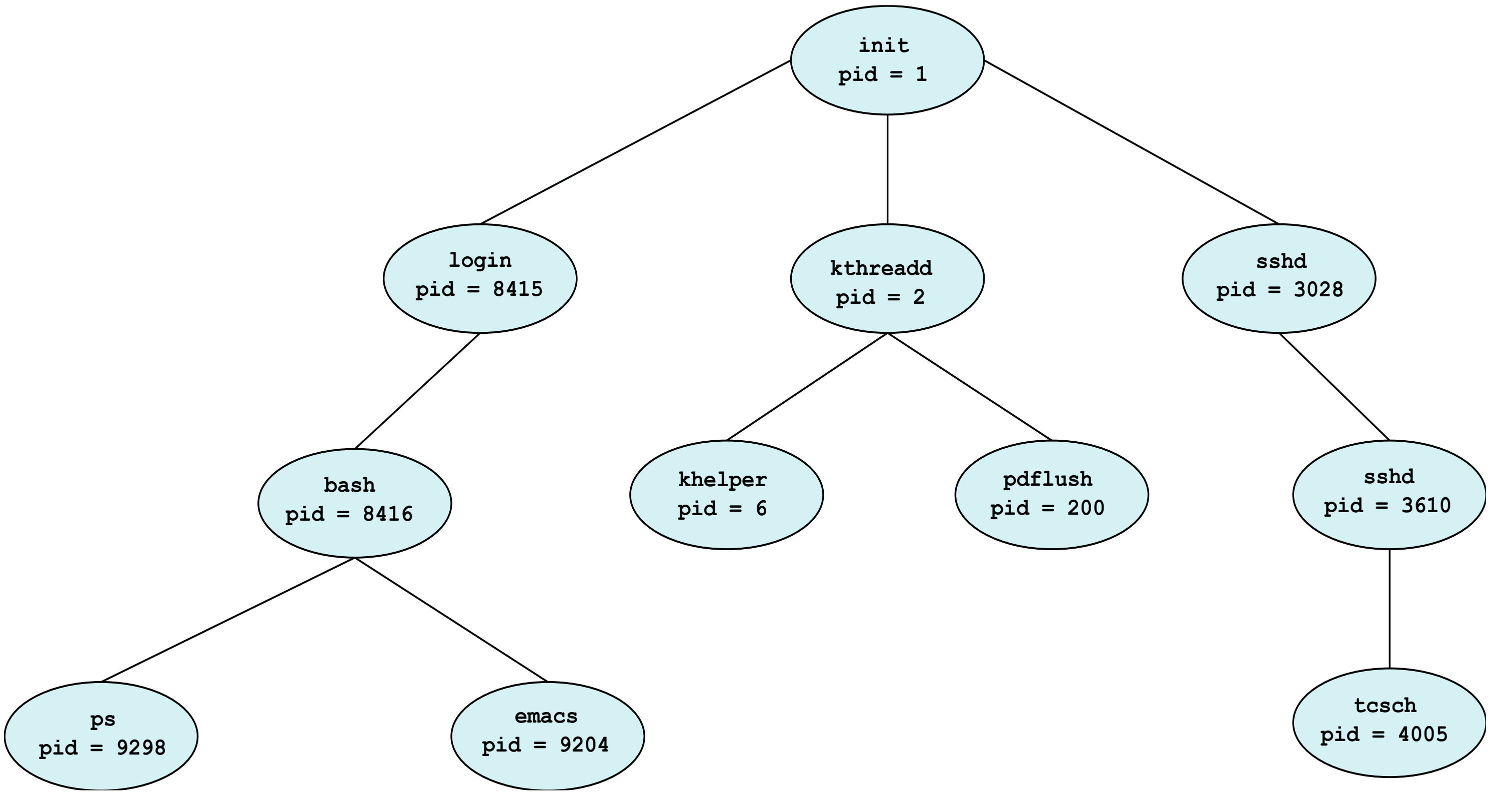
process termination,

and so on as detailed next

Process Creation

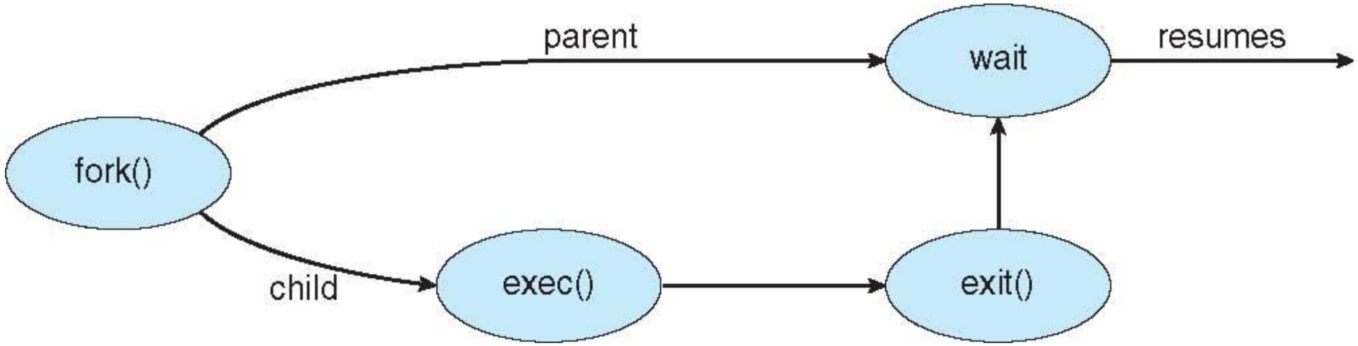
- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via a **process identifier** (**pid**)
- Resource sharing options
  - Parent and children share all resources
  - Children share subset 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

```
#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;
}
```

```
int main()
{
        pid_t  pid;
         /* fork another process */
         pid = fork();
         if (pid < 0) { /* error occurred */
                  fprintf(stderr, "Fork Failed");
                  exit(-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");
                  exit(0);
         }
}
```

parent process memory

child process memory

Creating a Separate Process via Windows API

```
#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**

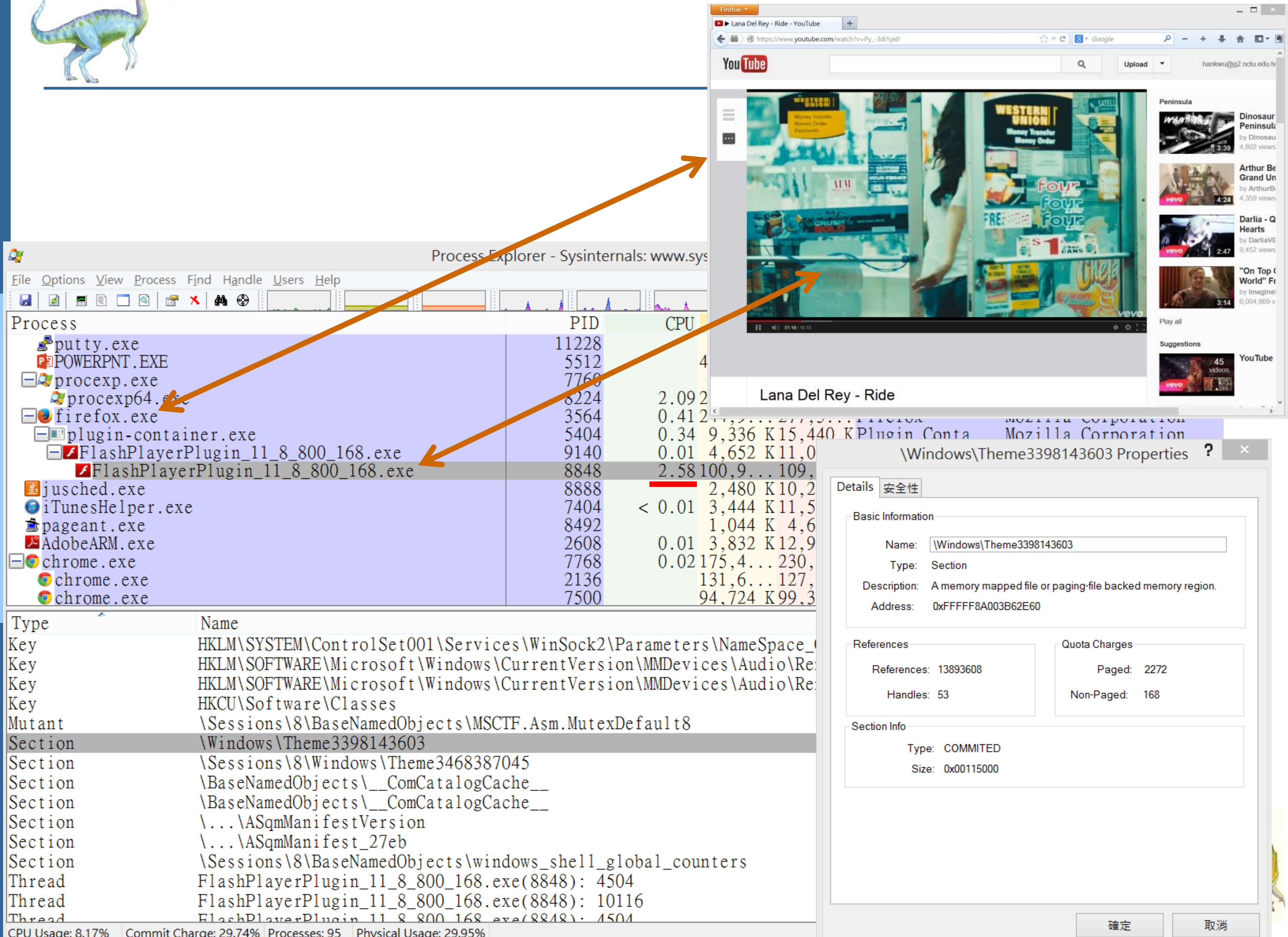
Multiprocess Architecture – Chrome Browser

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-in



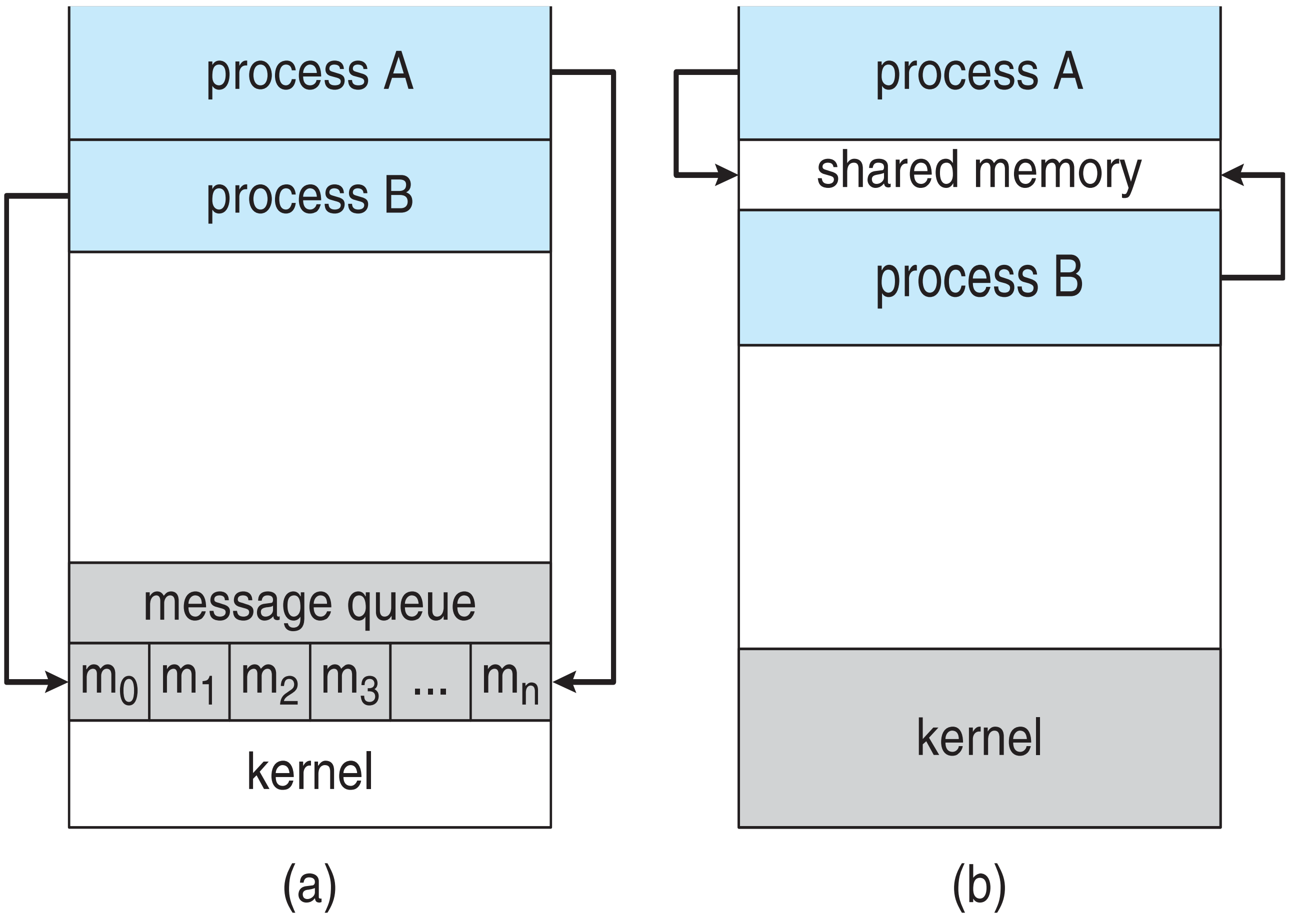
Process Explorer - Sysinternals: www.sys
File Options View Process Find Handle Users Help
Process PID CPU
putty.exe 11228
POWERPNT.EXE 5512
procexp.exe 7760
procexp64.exe 8224 2.09
firefox.exe 3564 0.41
plugin-container.exe 5404 0.34 9,336 K 15,440 K Plugin Conta Mozilla Corporation
FlashPlayerPlugin_11_8_800_168.exe 9140 0.01 4,652 K 11,0
FlashPlayerPlugin_11_8_800_168.exe 8848 2.58 100,9... 109,
jusched.exe 8888 2,480 K 10,2
iTunesHelper.exe 7404 < 0.01 3,444 K 11,5
pageant.exe 8492 1,044 K 4,6
AdobeARM.exe 2608 0.01 3,832 K 12,9
chrome.exe 7768 0.02 175,4... 230,
chrome.exe 2136 131,6... 127,
chrome.exe 7500 94,724 K 99,3
Type Name
Key HKLM\SYSTEM\ControlSet001\Services\WinSock2\Parameters\NameSpace_
Key HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\MMDevices\Audio\Re
Key HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\MMDevices\Audio\Re
Key HKCU\Software\Classes
Mutant \Sessions\8\BaseNamedObjects\MSCTF.Asm.MutexDefault8
Section \Windows\Theme3398143603
Section \Sessions\8\Windows\Theme3468387045
Section \BaseNamedObjects__ComCatalogCache__
Section \BaseNamedObjects__ComCatalogCache__
Section \...\ASqmManifestVersion
Section \...\ASqmManifest_27eb
Section \Sessions\8\BaseNamedObjects\windows_shell_global_counters
Thread FlashPlayerPlugin_11_8_800_168.exe(8848): 4504
Thread FlashPlayerPlugin_11_8_800_168.exe(8848): 10116
CPU Usage: 8.17% Commit Charge: 29.74% Processes: 95 Physical Usage: 29.95%
Lana Del Rey - Ride - YouTube
https://www.youtube.com/watch?v=Py_-3di1yx0
YouTube
Upload
Lana Del Rey - Ride
Peninsula
Play all
Suggestions
\Windows\Theme3398143603 Properties
Details 安全性
Basic Information
Name: \Windows\Theme3398143603
Type: Section
Description: A memory mapped file or paging-file backed memory region.
Address: 0xFFFFF8A003B62E60
References
References: 13893608
Handles: 53
Quota Charges
Paged: 2272
Non-Paged: 168
Section Info
Type: COMMITED
Size: 0x00115000
確定 取消

Interprocess Communication

- Processes within a system may be ***independent*** or ***cooperating***
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
  - **Shared memory**
  - **Message passing**

Communications Models

(a) Message passing.   (b) shared memory.



Cooperating Processes

Independent process cannot affect or be affected by the execution of another process

Cooperating process can affect or be affected by the execution of another process

Advantages of process cooperation

- Information sharing
- Computation speed-up
- Modularity
- Convenience

Producer-Consumer Problem

Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process

- **unbounded-buffer** places no practical limit on the size of the buffer
- **bounded-buffer** assumes that there is a fixed buffer size

Bounded-Buffer – Shared-Memory Solution

Shared data

```
#define BUFFER_SIZE 10
typedef struct {
  . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

Solution is correct, but can only use BUFFER_SIZE-1 elements

Bounded-Buffer – Producer

```
item next_produced;
while (true) {
        /* produce an item in next produced */
        while (((in + 1) % BUFFER_SIZE) == out)
                ; /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
}
```

Bounded Buffer – Consumer

```
item next_consumed;
while (true) {
        while (in == out)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;

        /* consume the item in next consumed */
}
```

Interprocess Communication – Shared Memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.
- Synchronization is discussed in great details in Chapter 5.

Interprocess Communication – Message Passing

Mechanism for processes to communicate and to synchronize their actions

Message system – processes communicate with each other without resorting to shared variables

IPC facility provides two operations:

`send`(*message*)

`receive`(*message*)

The *message* size is either fixed or variable

Message Passing (Cont.)

- If processes *P* and *Q* wish to communicate, they need to:
  - Establish a ***communication link*** between them
  - Exchange messages via send/receive
- Implementation issues:
  - How are links established?
  - Can a link be associated with more than two processes?
  - How many links can there be between every pair of communicating processes?
  - What is the capacity of a link?
  - Is the size of a message that the link can accommodate fixed or variable?
  - Is a link unidirectional or bi-directional?

Message Passing (Cont.)

- Implementation of communication link
  - Physical:
    - Shared memory
    - Hardware bus
    - Network
  - Logical:
    - Direct or indirect
    - Synchronous or asynchronous
    - Automatic or explicit buffering

Direct Communication

Processes must name each other explicitly:

- **`send`** (*P, message*) – send a message to process P
- **`receive`**(*Q, message*) – receive a message from process Q

Properties of communication link

- Links are established automatically
- A link is associated with exactly one pair of communicating processes
- Between each pair there exists exactly one link
- The link may be unidirectional, but is usually bi-directional

Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional

Indirect Communication

Operations

- create a new mailbox (port)
- send and receive messages through mailbox
- destroy a mailbox

Primitives are defined as:

`send`(*A, message*) – send a message to mailbox A

`receive`(*A, message*) – receive a message from mailbox A

Indirect Communication

- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends; P_2 and P_3 receive
  - Who gets the message?
- Solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send** -- the sender is blocked until the message is received
  - **Blocking receive** -- the receiver is blocked until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking send** -- the sender sends the message and continue
  - **Non-blocking receive** -- the receiver receives:
    - A valid message, or
    - Null message
- Different combinations possible
  - If both send and receive are blocking, we have a **rendezvous**

Synchronization (Cont.)

Producer-consumer becomes trivial

```
message next_produced;
while (true) {
     /* produce an item in next produced */
send(next_produced);
}
```

```
message next_consumed;
while (true) {
   receive(next_consumed);

   /* consume the item in next consumed */
}
```

Buffering

Queue of messages attached to the link.

implemented in one of three ways

1. Zero capacity – no messages are queued on a link.
   Sender must wait for receiver (rendezvous)
2. Bounded capacity – finite length of *n* messages
   Sender must wait if link full
3. Unbounded capacity – infinite length
   Sender never waits

Examples of IPC Systems - POSIX

POSIX Shared Memory

- Process first creates shared memory segment

```
shm_fd = shm_open(name, O CREAT | O RDWR, 0666);
```

- Also used to open an existing segment to share it
- Set the size of the object

```
ftruncate(shm fd, 4096);
```

- Now the process could write to the shared memory

```
sprintf(shared memory, "Writing to shared
memory");
```

IPC POSIX Producer

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* strings written to shared memory */
const char *message_0 = "Hello";
const char *message_1 = "World!";

/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

   /* create the shared memory object */
   shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

   /* configure the size of the shared memory object */
   ftruncate(shm_fd, SIZE);

   /* memory map the shared memory object */
   ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);

   /* write to the shared memory object */
   sprintf(ptr,"%s",message_0);
   ptr += strlen(message_0);
   sprintf(ptr,"%s",message_1);
   ptr += strlen(message_1);

   return 0;
}
```

IPC POSIX Consumer

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

   /* open the shared memory object */
   shm_fd = shm_open(name, O_RDONLY, 0666);

   /* memory map the shared memory object */
   ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

   /* read from the shared memory object */
   printf("%s",(char *)ptr);

   /* remove the shared memory object */
   shm_unlink(name);

   return 0;
}
```

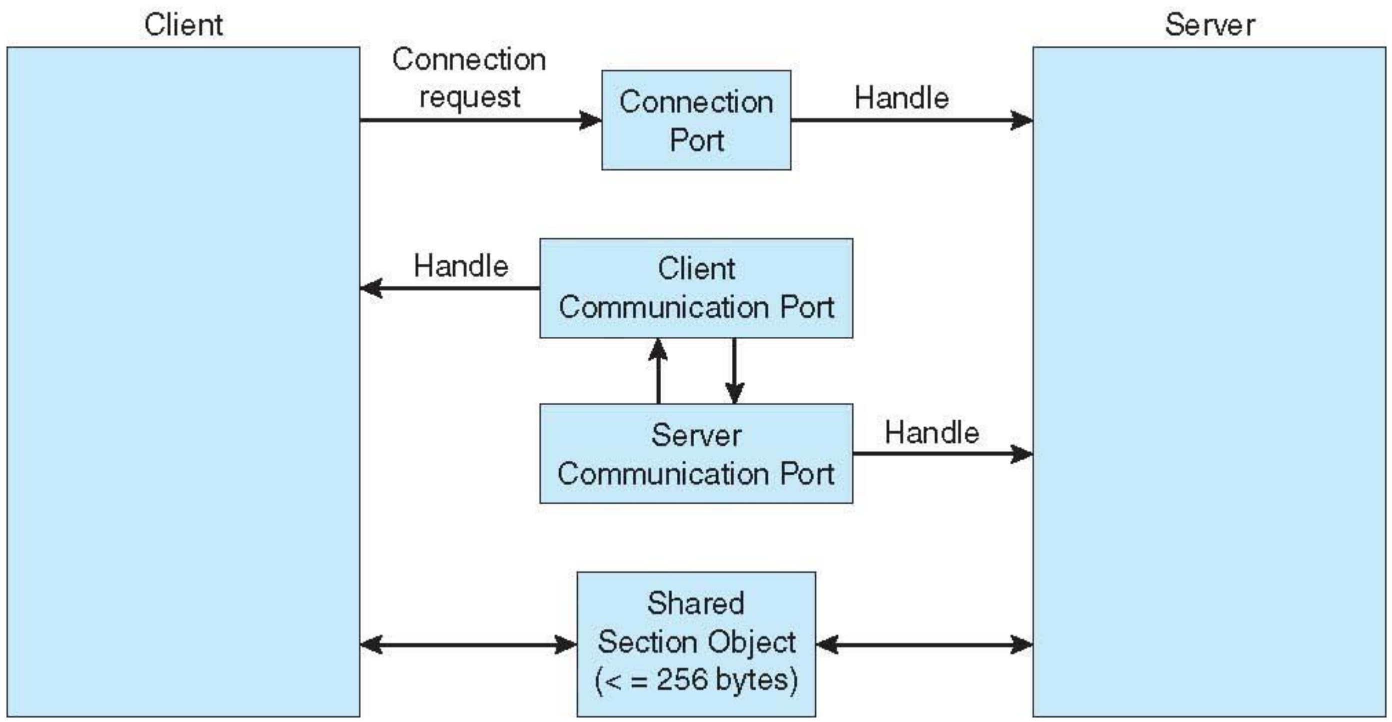
Examples of IPC Systems - Mach

- Mach communication is message based
  - Even system calls are messages
  - Each task gets two mailboxes at creation- Kernel and Notify
  - Only three system calls needed for message transfer
  - **`msg_send(), msg_receive(), msg_rpc()`**
  - Mailboxes needed for commuication, created via
  - **`port_allocate()`**
  - Send and receive are flexible, for example four options if mailbox full:
    - Wait indefinitely
    - Wait at most n milliseconds
    - Return immediately
    - Temporarily cache a message

Examples of IPC Systems – Windows

- Message-passing centric via **advanced local procedure call (LPC)** facility
  - Only works between processes on the same system
  - Uses ports (like mailboxes) to establish and maintain communication channels
  - Communication works as follows:
    - The client opens a handle to the subsystem's **connection port** object.
    - The client sends a connection request.
    - The server creates two private **communication ports** and returns the handle to one of them to the client.
    - The client and server use the corresponding port handle to send messages or callbacks and to listen for replies.

Local Procedure Calls in Windows



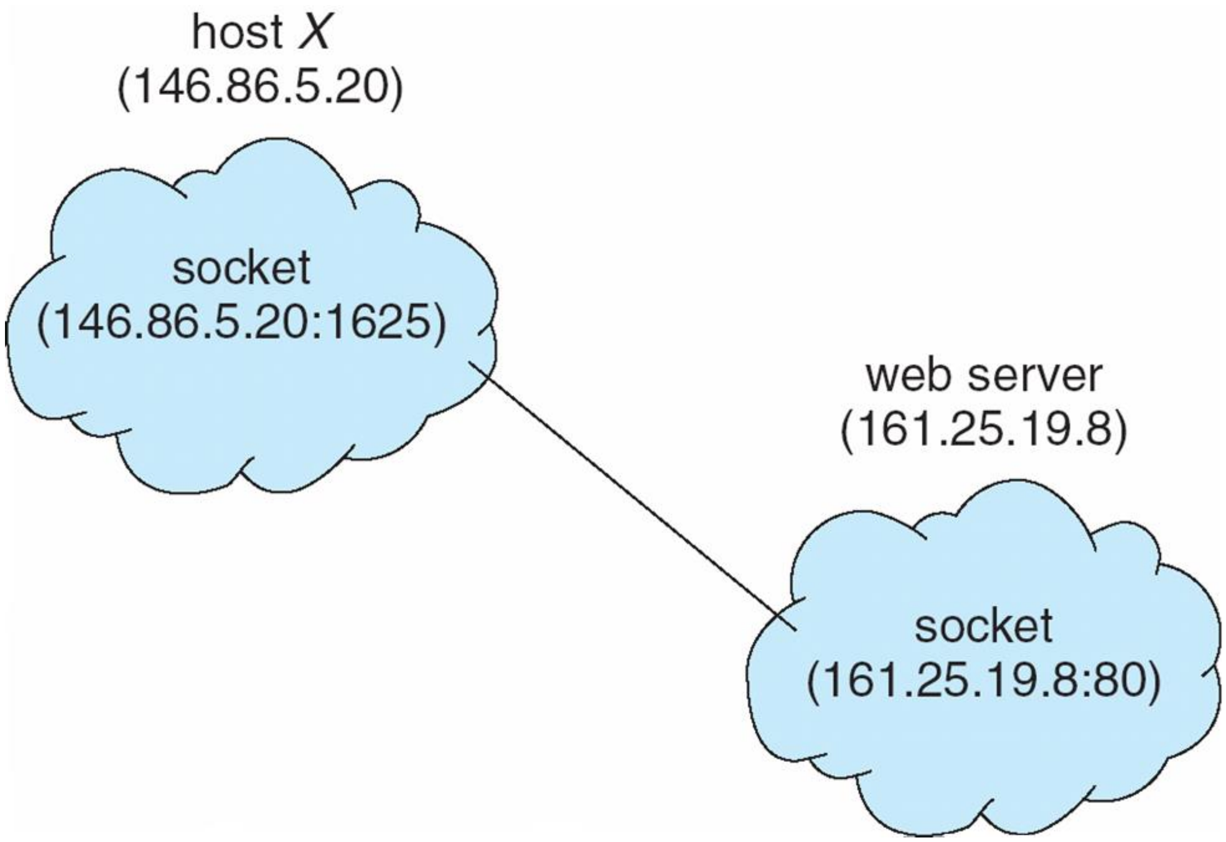
Communications in Client-Server Systems

- Sockets
- Remote Procedure Calls
- Pipes
- Remote Method Invocation (Java)

Sockets

- A **socket** is defined as an endpoint for communication
- Concatenation of IP address and **port** – a number included at start of message packet to differentiate network services on a host
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- Communication consists between a pair of sockets
- All ports below 1024 are ***well known***, used for standard services
- Special IP address 127.0.0.1 (**loopback**) to refer to system on which process is running

Socket Communication



Sockets in Java

Three types of sockets

- **Connection-oriented** (**TCP**)
- **Connectionless** (**UDP**)
- **`MulticastSocket`** class– data can be sent to multiple recipients

Consider this "Date" server:

```
import java.net.*;
import java.io.*;

public class DateServer
{
   public static void main(String[] args) {
      try {
         ServerSocket sock = new ServerSocket(6013);

         /* now listen for connections */
         while (true) {
            Socket client = sock.accept();

            PrintWriter pout = new
             PrintWriter(client.getOutputStream(), true);

            /* write the Date to the socket */
            pout.println(new java.util.Date().toString());

            /* close the socket and resume */
            /* listening for connections */
            client.close();
         }
      }
      catch (IOException ioe) {
         System.err.println(ioe);
      }
   }
}
```

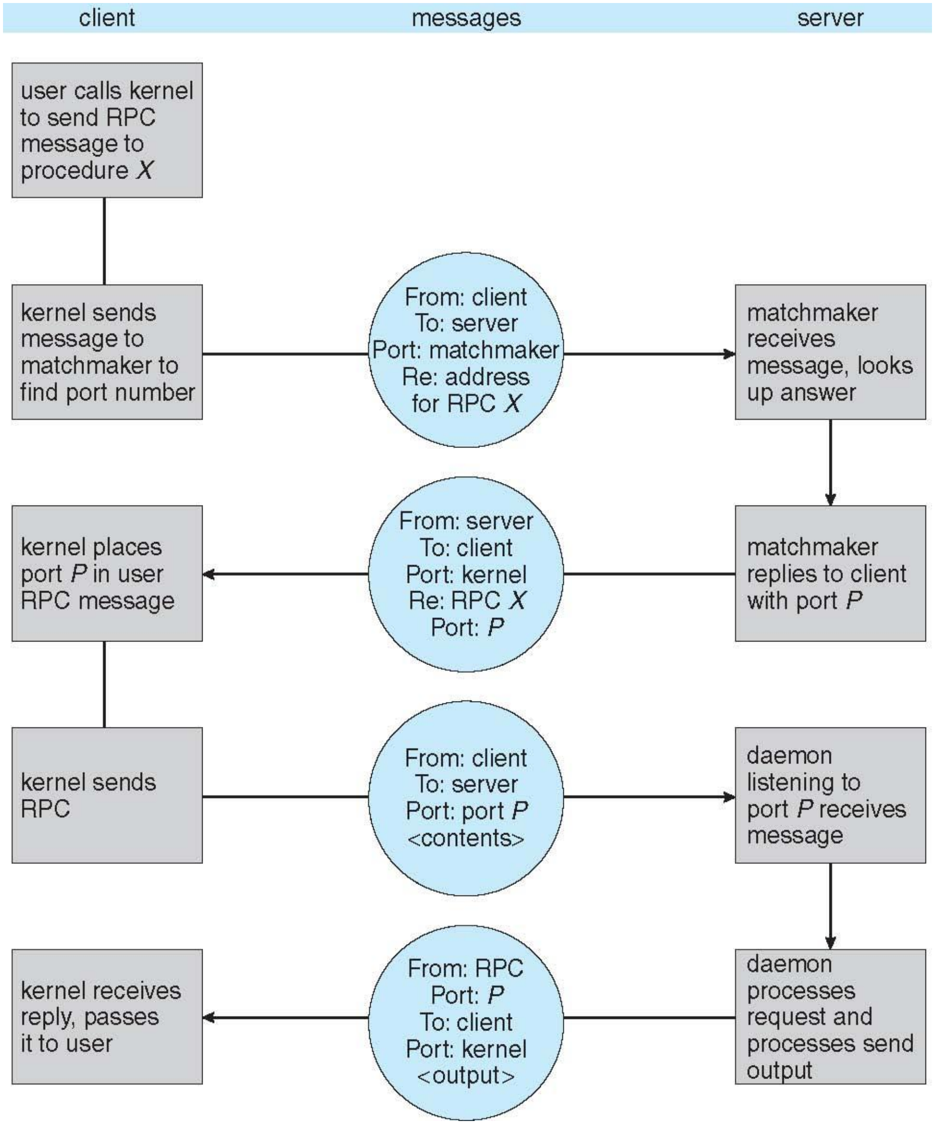
Remote Procedure Calls

- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems
  - Again uses ports for service differentiation
- **Stubs** – client-side proxy for the actual procedure on the server
- The client-side stub locates the server and **marshalls** the parameters
- The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server
- On Windows, stub code compile from specification written in **Microsoft Interface Definition Language** (**MIDL**)

Remote Procedure Calls (Cont.)

- Data representation handled via **External Data Representation** (**XDL**) format to account for different architectures
  - **Big-endian** and **little-endian**
- Remote communication has more failure scenarios than local
  - Messages can be delivered ***exactly once*** rather than ***at most once***
- OS typically provides a rendezvous (or **matchmaker**) service to connect client and server

Execution of RPC



Pipes

- Acts as a conduit allowing two processes to communicate
- Issues:
  - Is communication unidirectional or bidirectional?
  - In the case of two-way communication, is it half or full-duplex?
  - Must there exist a relationship (i.e., ***parent-child***) between the communicating processes?
  - Can the pipes be used over a network?
- Ordinary pipes – cannot be accessed from outside the process that created it. Typically, a parent process creates a pipe and uses it to communicate with a child process that it created.
- Named pipes – can be accessed without a parent-child relationship.

Ordinary Pipes

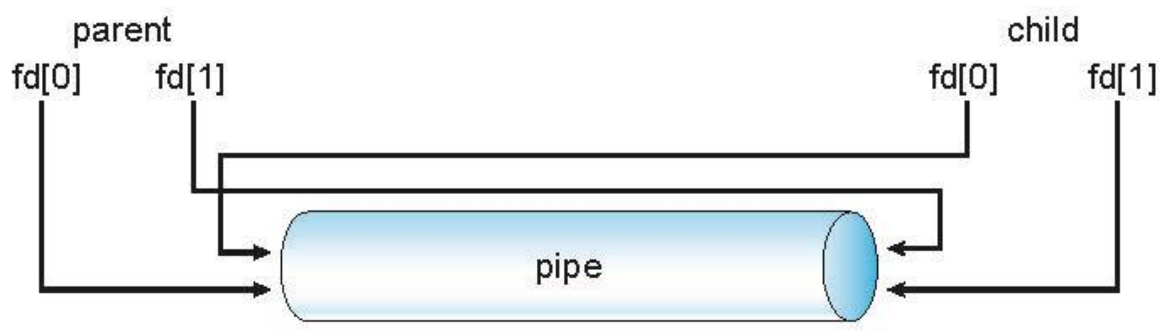
Ordinary Pipes allow communication in standard producer-consumer style

Producer writes to one end (the **write-end** of the pipe)

Consumer reads from the other end (the **read-end** of the pipe)

Ordinary pipes are therefore unidirectional

Require parent-child relationship between communicating processes



Windows calls these **anonymous pipes**

See Unix and Windows code samples in textbook

Named Pipes

- Named Pipes are more powerful than ordinary pipes
- Communication is bidirectional
- No parent-child relationship is necessary between the communicating processes
- Several processes can use the named pipe for communication
- Provided on both UNIX and Windows systems

End of Chapter 3