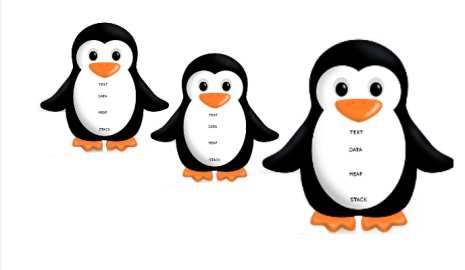


Process Management



Learning Objectives

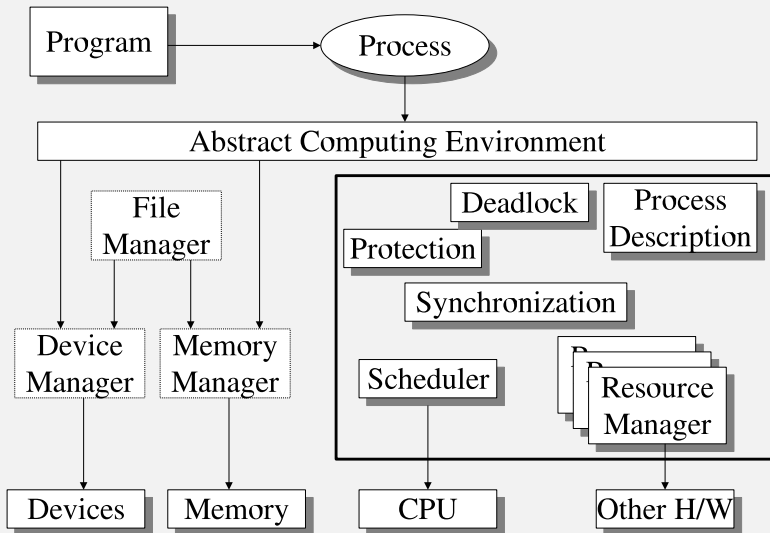
- ▶ Understand how executables are structured, loaded and run
- ▶ Understand the memory hierarchy as related to processes
- ▶ Explain the process abstraction and its implementation in the Operating System
- ▶ Understand process state using state diagrams

System View of a Process

The **process manager** implements the process abstraction. It covers the following areas:

- ▶ Scheduling of processes on the CPU(s)
- ▶ Synchronization mechanisms for processes
- ▶ Responsible for dealing with deadlocks among processes
- ▶ Partially responsible for protection and security

Process Manager Overview



Process Address Space

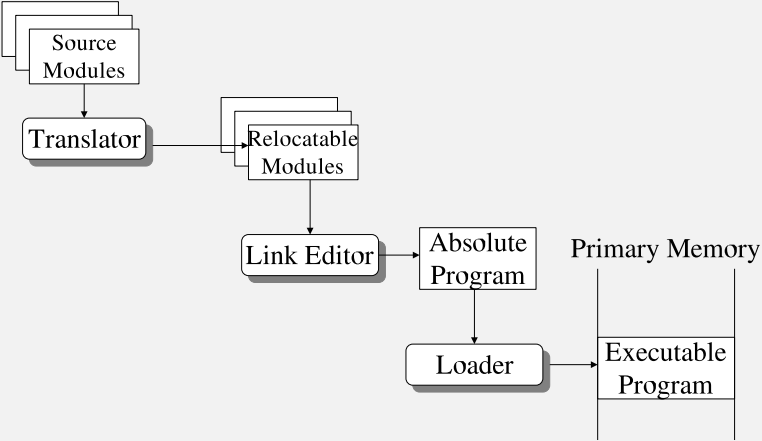
program & libraries $\xrightarrow{\text{compile/link}}$ executable $\xrightarrow{\text{load}}$ process

- ▶ A program is a set of source code modules that reference each other and reference a collection of library object modules
- ▶ The *address space* is a set of linearly ordered locations used by the process to reference program text, data, operating system services, resources etc
- ▶ A program *image* defines the set of all primary memory addresses a process uses

Generating the Address Space

- ▶ Compiling and linking produces an *absolute program* (`a.out`).
- ▶ The loader maps the address space to the allocated primary memory addresses and sets the PC (program counter) to the first executable instruction (a.k.a. *main entry point*).

Process Management



The Structure of Executable Files

- ▶ The structure of an executable file is dependent upon the operating system
- ▶ The compiler/linker needs to produce a file in one of the formats understood by the operating system to be executable
- ▶ Older standard executable formats from Unix: **COFF** (Common Object File Format) and **a.out**
 - ▶ Linux and most modern Unixes use the **ELF** (Executable and Linkable Format) format
 - ▶ MS Windows uses the **PE** (Portable Executable, derived from the COFF format) format
 - ▶ MAC OS X uses the **Mach-O** format (derived from the a.out format)

The ELF Executable format

- ▶ Flexible and extensible, not bound to any particular processor or architecture
- ▶ Each ELF file is made up of one ELF header that describes the layout of the file. Then follow physical (or program) headers that describe the program segments. These include the *text* segment (compiled code), *read only data*, *data* segment (initialized global and static variables) and others.
- ▶ The executable image on the disk does not set aside space for uninitialized data segment variables. The uninitialized part of the data segment is set to zero after being loaded into memory. The section that stores these types of variables is called the **BSS** (Block Started by a Symbol) section.
 - ▶ Example: `test-bss.c`
 - ▶ What is the advantage of having a BSS section?
- ▶ ELF uses position independent code and a global offset table, which trades off execution time against memory usage in favor of the latter



Process Management

e_ident	'E' 'L' 'F'
e_entry	0x8048090
e_phoff	52
e_phentsize	32
e_phnum	2
...	...
p_type	PT_LOAD
p_offset	0
p_vaddr	0x8048000
p_filesz	68532
p_memsz	68532
p_flags	PF_R, PF_X
p_type	PT_LOAD
p_offset	68536
p_vaddr	0x8059BB8
p_filesz	2200
p_memsz	4248
p_flags	PF_R, PF_W
Code	
Data	

ELF Executable Image (statically linked file)

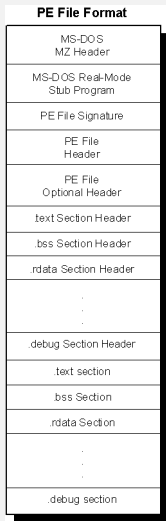
ELF layout

- ▶ How to look at executables: one way is to use the dump command `od`. It has various options to examine the data in hex, octal, ASCII etc. Under KDE there is a utility called `okteta`, which is a nice GUI hex editor
- ▶ Use the utility `readelf` to peek into the structure of an ELF file.
- ▶ Look at the header file `/usr/include/linux/elf.h` for more details of the ELF executable format.
- ▶ See example `process-management/display-elf-headers.c` for a program that reads headers from an ELF executable file.
- ▶ Dynamically linked executables have more segments due to linking with libraries.

Microsoft's Portable Executable (PE) Format

- ▶ Based on the COFF (Common Object File Format from Unix). Retains the old MZ header from MS-DOS to remain backwards compatible. Works on all Windows operating systems since NT 3.1
- ▶ Consists of an MS-DOS MZ header, followed by a real-mode stub program, the PE file signature, the PE file header, the PE optional header, all of the section headers, and finally, all of the section bodies
- ▶ The PE file format has eleven predefined sections, as is common to applications for Windows API, but each application can define its own unique sections for code and data
- ▶ The `.debug` predefined section also has the capability of being stripped from the file into a separate debug file. If so, a special debug header is used to parse the debug file, and a flag is specified in the PE file header to indicate that the debug data has been stripped

Microsoft's Portable Executable Format Layout



Source for image: <http://www.csn.ul.ie/~caolan/publink/winresdump/winresdump/doc/pefile.html>

Microsoft's Portable Executable (PE) Format

- ▶ The .NET framework uses an extended PE file format. There is another extension known as PE32+ (or PE+) for 64-bit systems as well as one for the embedded Windows CE system
- ▶ PE files use a preferred base address and all addresses generated by the compiler/linker are fixed ahead of time to speed up execution. However, if the preferred base address isn't available, then an expensive "rebasng" operation must be done that can result in having to copy shared libraries and causing a loss of memory efficiency

Consistency in the Address Space

- ▶ **Memory Hierarchy:**

Registers \leftarrow Primary Memory \leftarrow Secondary Memory.

Registers \leftarrow on-chip Cache \leftarrow off-chip Cache \leftarrow Primary Memory \leftarrow Secondary Memory

- ▶ The memory hierarchy is consistent for locations that contain instructions (since programs are not allowed to be self-modifying). But the data values are not consistent unless the programmer explicitly makes them consistent
- ▶ For a given variable, we have its value in a register (M_{R_i}), its value in the primary memory (M_{P_j}), and its value in secondary memory (M_{S_k})
- ▶ What happens when a CPU is switched to another process?
What happens when memory manager deallocates some of the space used by a process?
- ▶ Linux system calls to synchronize memory images with disk images:
`fdatasync` - synchronize a file's in-core data with that on disk
`sync` - synchronize a file's complete in-core state with that on disk



The Process Descriptor

The **process descriptor** is the primary data structure used to keep track of the status of a process and the specific environment that is associated with a process. It contains the following types of information:

- ▶ process state (whether it is blocked or ready)
- ▶ memory state
- ▶ current processor register contents
- ▶ pointer to the stack for the process
- ▶ resources (those allocated and those waiting for)
- ▶ other information

Process Descriptor

- ▶ A process descriptor is allocated when process is created and deallocated when a process dies. Usually there is a limit on the number of process descriptors in an operating system.
- ▶ Even though the process manager is the one primarily interacting with the process descriptor it is also queried and some fields are modified by other parts of the operating system
- ▶ *How to find the process descriptor in Linux source code?*
Start with the source code for `fork()` system call (in the file `kernel/fork.c`) The obvious candidate is the structure `task_struct`, which is found in the header file `include/linux/sched.h` in the kernel source.
(Use `grep "task_struct {" *.h` in the directory `include/linux` in the kernel source code)

Linux Processes

In Linux terminology, they are called *tasks*. Linux has a list of process descriptors (which are of type `task_struct` defined in `sched.h` in your Linux kernel source tree)

- ▶ The maximum number of threads/processes allowed is dependent upon the amount of memory in the system. Check `/proc/sys/kernel/threads-max` for the current limit.
- ▶ By writing to that file, the limit can be changed on the fly (by the superuser). Or set it in `/etc/sysctl.conf` to set it at bootup time.
- ▶ There is also a limit on max pid to be 32768 (2^{15}) to make 2.6 and newer kernels compatible with programs written for the older kernels. This limit can be seen in `/proc/sys/kernel/pid_max`
- ▶ This can be overwritten to any value up to 2^{22} (about 4 million). For example:

```
echo 1000000 > /proc/sys/kernel/pid_max
```

To do it permanently, add
`kernel.pid_max = 1000000`
to the `/etc/sysctl.conf` file so it gets set at bootup time.
- ▶ Look at `include/linux/threads.h` in kernel source code to see the limits.

Linux Process Descriptor

Browse live in the file `include/linux/sched.h` in the kernel source...some snippets given below.

```
struct task_struct {
    volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
    void *stack;
    atomic_t usage;
    unsigned int flags; /* per process flags, defined below */
    unsigned int ptrace;
    ...
    int prio, static_prio, normal_prio;
    unsigned int rt_priority;
    const struct sched_class *sched_class;
    struct sched_entity se;
    struct sched_rt_entity rt;
    ...
    struct mm_struct *mm, *active_mm;
    int exit_state;
    int exit_code, exit_signal;
    int pdeath_signal; /* The signal sent when the parent dies */
    ...
    pid_t pid;
    pid_t tgid;
```

Linux Process Descriptor (contd.)

```
/* pointers to (original) parent process, youngest child, younger sibling,
 * older sibling, respectively. (p->father can be replaced with p->real_parent->pid)
 */
struct task_struct *real_parent; /* real parent process */
struct task_struct *parent; /* recipient of SIGCHLD, wait4() reports */
/* children/sibling forms the list of my natural children */
struct list_head children; /* list of my children */
struct list_head sibling; /* linkage in my parent's children list */
struct task_struct *group_leader; /* threadgroup leader */
...
/* PID/PID hash table linkage. */
struct pid_link pids[PIDTYPE_MAX];
struct list_head thread_group;
...
cputime_t utime, stime, utimescaled, stimescaled;
cputime_t gtime;
...
/* CPU-specific state of this task */
struct thread_struct thread;
/* filesystem information */
struct fs_struct *fs;
/* open file information */
struct files_struct *files;
/* signal handlers */
struct signal_struct *signal;
struct sighand_struct *sighand;
sigset_t blocked, real_blocked;
sigset_t saved_sigmask; /* restored if set_restore_sigmask() was used */
struct sigpending pending;
...
};
```

Data Structures for Processes (1)

What data structure(s) are used to keep track of the processes?

- ▶ Linux kernel: The process descriptors are kept in circular linked lists, a binarization of a general tree data structure and a hash table simultaneously!



- ▶ **In-class Exercise.** Sketch a sample declaration of a general tree using an array of child pointers (with **MAX**, say 100, children per node).
- ▶ **In-class Exercise.** Sketch a sample declaration of *binarization* of a general tree with the **leftmost-child-right-sibling** representation.
- ▶ **In-class Exercise.** How much space is wasted (as null pointers) in a n node tree represented in the above two layouts?

Data Structures for Processes (2)

- ▶ The linked list implementation in the kernel uses the following node (check `include/linux/types.h` and `include/linux/list.h`):

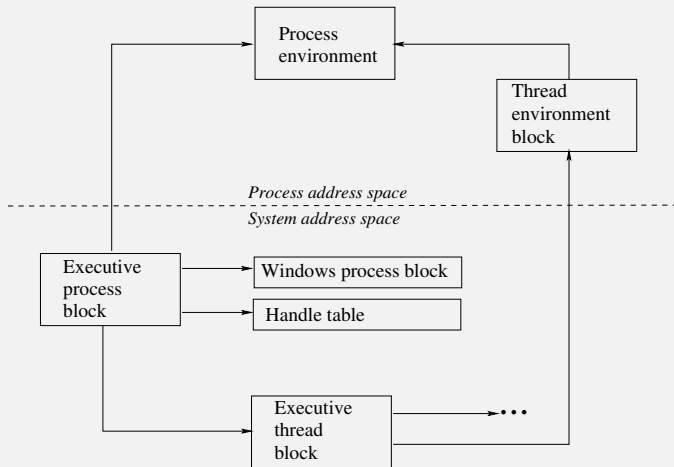
```
struct list_head {
    struct list_head *next, *prev;
};
```

- ▶ The linked lists are circular, so there is no head or tail node. We can traverse the whole list starting from any node.

MS Windows Processes

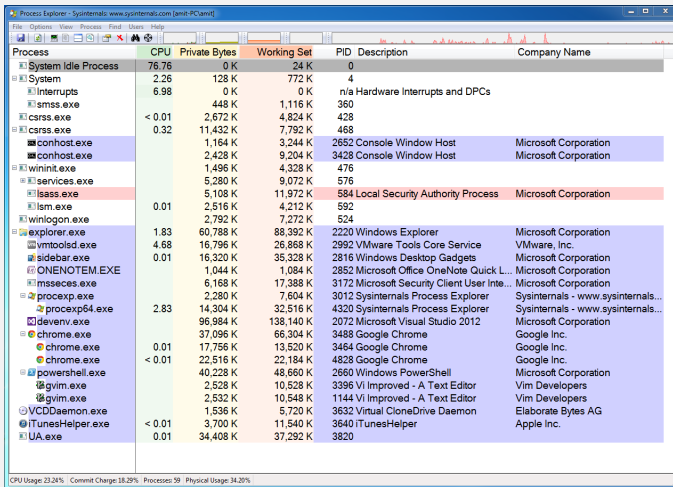
- ▶ Processes and Threads are kept track of in separate data structures.
- ▶ Each windows process is represented by a *executive process block*. It contains pointers to other structures, including *executive thread blocks*. Part of the information is stored in the *process environment block* so it can be accessed in user space
- ▶ The windows subsystem process (Csrss: client/server run time subsystem) maintains a parallel data structure for each process that is executing a Windows program. The kernel-mode part of the Windows subsystem (Win32k.sys) also maintains a per-process data structure
- ▶ Compare with the Linux approach of representing processes and threads both with a task structure

MS Windows process/thread data structures



MS Windows process tree

Process Management



Process	CPU	Private Bytes	Working Set	PID	Description	Company Name
System Idle Process	76.76	0 K	24 K	0		
System	2.26	128 K	772 K	4		
Interrupts	6.98	0 K	0 K	n/a	Hardware Interrupts and DPCs	
smss.exe		448 K	1,116 K	360		
csrss.exe	< 0.01	2,672 K	4,824 K	428		
csrss.exe	0.32	11,432 K	7,792 K	468		
conhost.exe		1,164 K	3,244 K	2652	Console Window Host	Microsoft Corporation
conhost.exe		2,428 K	9,204 K	3428	Console Window Host	Microsoft Corporation
wininit.exe		1,496 K	4,328 K	476		
services.exe		5,280 K	9,072 K	576		
lsass.exe		5,108 K	11,972 K	584	Local Security Authority Process	Microsoft Corporation
lsm.exe	0.01	2,516 K	4,212 K	592		
winlogon.exe		2,792 K	7,272 K	524		
explorer.exe	1.83	60,788 K	88,392 K	2220	Windows Explorer	Microsoft Corporation
vmtoolsd.exe	4.68	16,796 K	26,868 K	2992	VMware Tools Core Service	VMware, Inc.
sidebar.exe	0.01	16,320 K	35,328 K	2816	Windows Desktop Gadgets	Microsoft Corporation
ONENOTEM.EXE		1,044 K	1,084 K	2852	Microsoft Office OneNote Quick L...	Microsoft Corporation
mssecss.exe		6,168 K	17,388 K	3172	Microsoft Security Client User Inte...	Microsoft Corporation
procexp.exe		2,280 K	7,604 K	3012	Sysinternals Process Explorer	Sysinternals - www.sysinternals...
procexp64.exe	2.83	14,304 K	32,516 K	4320	Sysinternals Process Explorer	Sysinternals - www.sysinternals...
devenv.exe		96,984 K	138,140 K	2072	Microsoft Visual Studio 2012	Microsoft Corporation
chrome.exe		37,096 K	66,304 K	3488	Google Chrome	Google Inc.
chrome.exe	0.01	17,756 K	13,520 K	3464	Google Chrome	Google Inc.
chrome.exe	< 0.01	22,516 K	22,184 K	4828	Google Chrome	Google Inc.
powershell.exe		40,228 K	48,660 K	2660	Windows PowerShell	Microsoft Corporation
gvim.exe		2,528 K	10,528 K	3396	Vi Improved - A Text Editor	Vim Developers
gvim.exe		2,532 K	10,548 K	1144	Vi Improved - A Text Editor	Vim Developers
VCDDaemon.exe		1,536 K	5,720 K	3632	Virtual CloneDrive Daemon	Elaborate Bytes AG
iTunesHelper.exe	< 0.01	3,700 K	11,540 K	3640	iTunesHelper	Apple Inc.
UA.exe	0.01	34,408 K	37,292 K	3820		

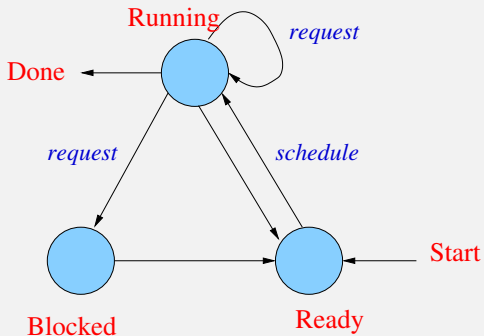
CPU Usage: 23.24% Commit Charge: 18.25% Processes: 59 Physical Usage: 34.25%

Screenshot of procexp program from Sysinternals tools.

Process State Diagram

- ▶ A **process state diagram** is used to characterize the behavior of a process
- ▶ A process may be *ready*, *running* or *blocked*. How does the process state change?
- ▶ The ready and blocked states can be refined to *Active* and *Suspended*

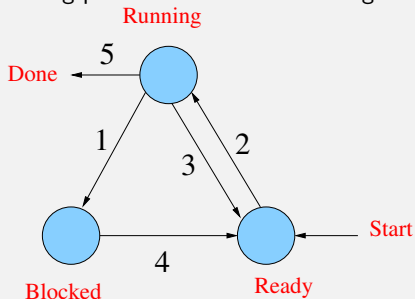
Simple Process State Diagram





In-class Exercise

Consider the following process state transition diagram:



For each of the transitions give an example of a specific event that can cause that transition.

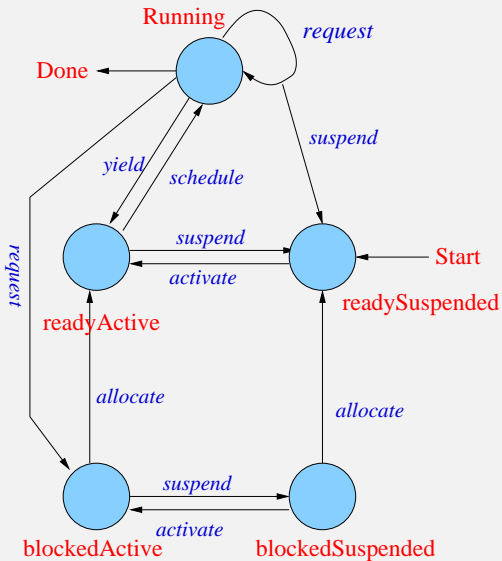
(1)

(2)

(3)

(4)

Extended Process State Diagram

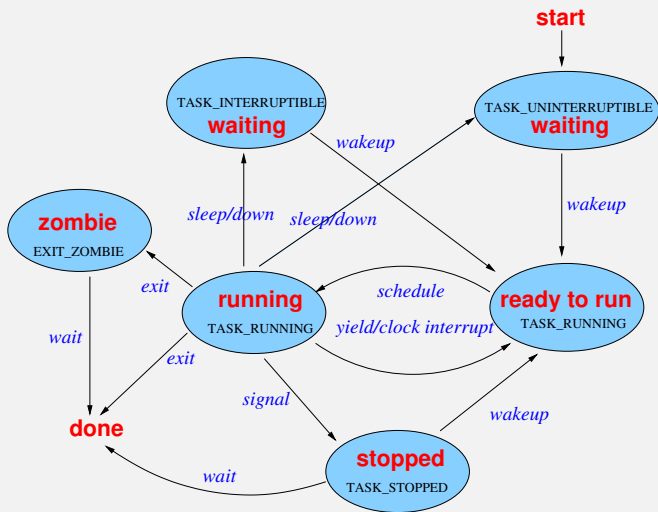




In-class Exercise

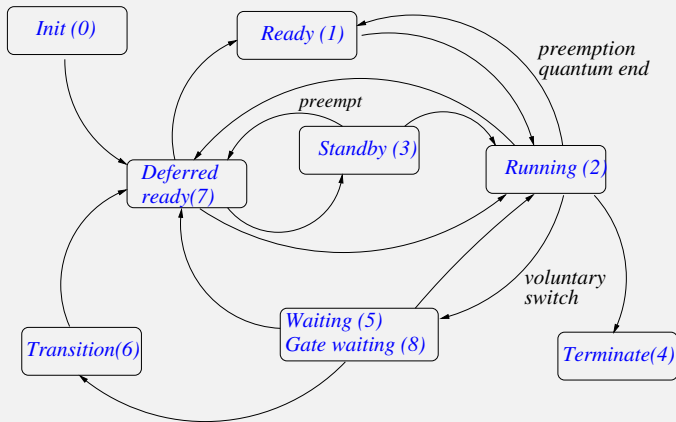
- ▶ Give two specific examples of how a process could be involuntarily removed from the CPU.
- ▶ Give two specific examples of how a process could voluntarily give up the CPU.
- ▶ Give a specific example of how a process could move from a running state to the *readySuspended* state.

Linux Process State Diagram



See the file `include/linux/sched.h` for the Linux process states.

MS Windows Thread/Process State Diagram



MS Windows process states

- ▶ **Init (0)**: Used internally while the thread is being created
- ▶ **Ready (1)**: A thread in the ready state is waiting to execute
- ▶ **Running (2)**: Running on a processor until the quantum ends, it is preempted, it terminates, it yields or it voluntarily enters the wait state
- ▶ **Standby(3)**: A thread in the standby state has been selected to run on a particular processor. Only one thread can be in a standby state for each processor on the system. Threads can be preempted from this state
- ▶ **Terminated (4)**: In the terminated state, the executive thread block might or might not be deallocated depending on the policy that is set

MS Windows process states (contd.)

- ▶ **Waiting (5)**: A thread can enter the waiting state in several ways: it can voluntarily wait for an object to synchronize its execution, the operating system can wait on its behalf (such as for paging I/O), or it can be suspended by an external entity
- ▶ **Transition (6)**: A thread is ready for execution but its kernel stack is paged out of memory
- ▶ **Deferred Ready (7)**: Threads that have been selected to run on a specific processor but have not yet been scheduled. This is so that the kernel can minimize the amount of time the system wide lock on the scheduling database is held
- ▶ **Gate Waiting (8)**: When thread does a wait on a gate dispatcher object (separate from the waiting state). Because gates don't use the dispatcher lock, but a per object lock, the kernel needs to be able to distinguish it if it has to break the lock

Resources

A process manager also manages resources used by processes.

- ▶ A **resource** is anything that can block a process from executing. Examples include: memory, messages, input data, disks, tapes, files etc
- ▶ A resource that can be allocated and must be returned after the process has finished are called **reusable resources**. These kind of resources are fixed in number. What if a process does not release reusable resources?
- ▶ A resource that is never released is called a **consumable resource**. These kind of resources are unbounded
- ▶ Different resource allocation strategies. Give more responsibilities to client processes. E.g. user level threads, IBM virtual machine operating system, virtual machine servers, client-server operating systems and microkernels