Introduction into Multicore Programming

Károly Bósa
(Karoly.Bosa@jku.at)

Research Institute for Symbolic Computation
(RISC)

Topic

Processes:

- What is a process?
- Why Processes and Not Threads?
- Process Control Block
- The address space/image of a process
- Process States
- Process Scheduling
- Context Switch
- Monitoring Processes with the ps Utility
- Creating Processes
- Calls Regarding Processes
- Introduction into Process Tracing

What Is a Process?

- A *process* is a unit of work created by the operating system:
    - it must have an address space assigned to it by the operating system.
    - It must have a process id.
    - It must have a state and an entry in the process table.
- A process has a set of executing instructions that resides in the address space of that process.
- It is important to note that processes and programs are not necessarily equivalent.

Why Processes and Not Threads?

- Threads turn out to be easier to program because threads share the same address space (communication and synchronization between threads much easier).

- But:
  - Processes have their own address space (provide a certain amount security and isolation).
  - For multiuser applications, each user's process must be isolated.
  - Operating system resources are assigned primarily to processes and then shared by threads.
  - The number of open files that threads may use is limited to how many open files a single process can have.

Type of Processes

- Processes that execute system code are called **system processes** ,or **kernel processes**.

- **User processes** execute their own code, and sometimes they make system function calls:

    - When a user process executes its own code, it is in **user mode**.

    - In **kernel mode**, a user process makes a system function call (for example, read() , write() , or open()), it is executing operating system instructions.

      If the processor is given to the kernel to complete the system call, it cannot be *preempted* by any user processes.

- **Remark:** *preemption* is the act of temporarily interrupting a task being carried out by the CPU, and with the intention of resuming the task at a later time.

Process Control Block I.

- Processes have characteristics that identify them and determine their behavior during execution.
- The kernel maintains data structures and provides system functions that allow the user to have access to this information.
- Some information is stored in the *process control block (**PCB**)* .
- PCB is needed for the operating system to manage each process.
- When the operating system switches between a process utilizing the CPU to another process, …
  - … it saves the current state of the executing process and its context to the PCB,
  - in order to restart the process the next time it is assigned to the CPU.

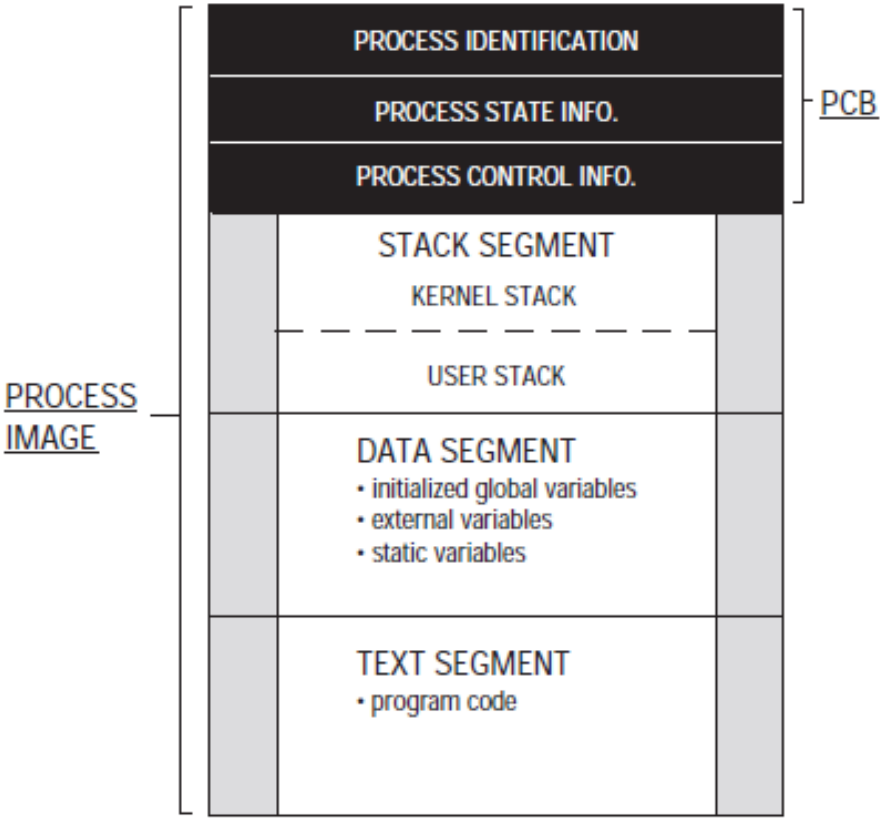
Process Control Block II.

- PCB information includes:

process control	–Current state and priority of the process –Pointers to allocated resources –Pointers to location of the process's memory –Pointer to the process's parent and child processes
state of the processor	–Processor utilized by process –Control and status registers –Stack pointers
process identification	–Process, parent, and child identifiers

The address space of a process I.

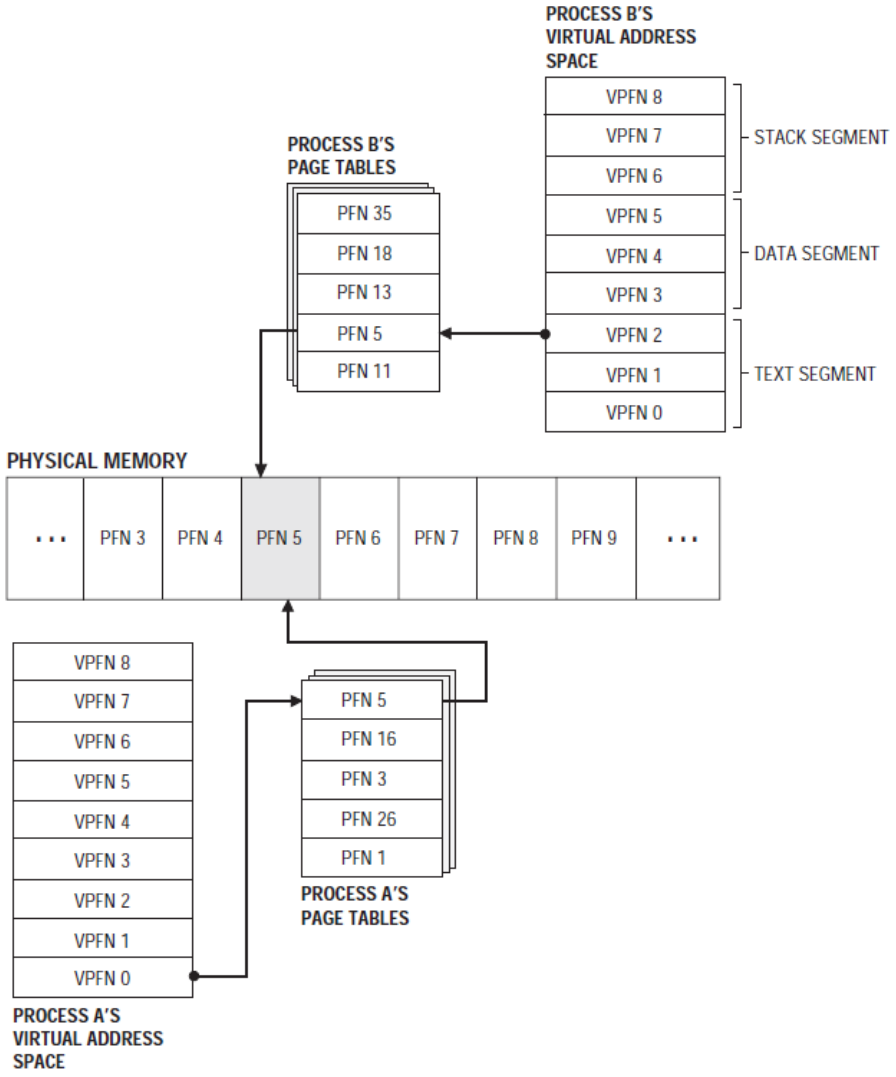
Karoly.Bosa@jku.at



The address space of a process II.

Karoly.Bosa@jku.at

- The address space of a process is *virtual.*
- The segments of the process's virtual address space are contiguous blocks of memory.
- Physical address space is broken up into chunks called *pages* . Each page has a unique *page frame number(PFN)*.
- Each segment is broken up *virtual pages.*
- The *virtual page frame number (VPFN)* is used as an index into the process's *page tables* which contain PFNs.

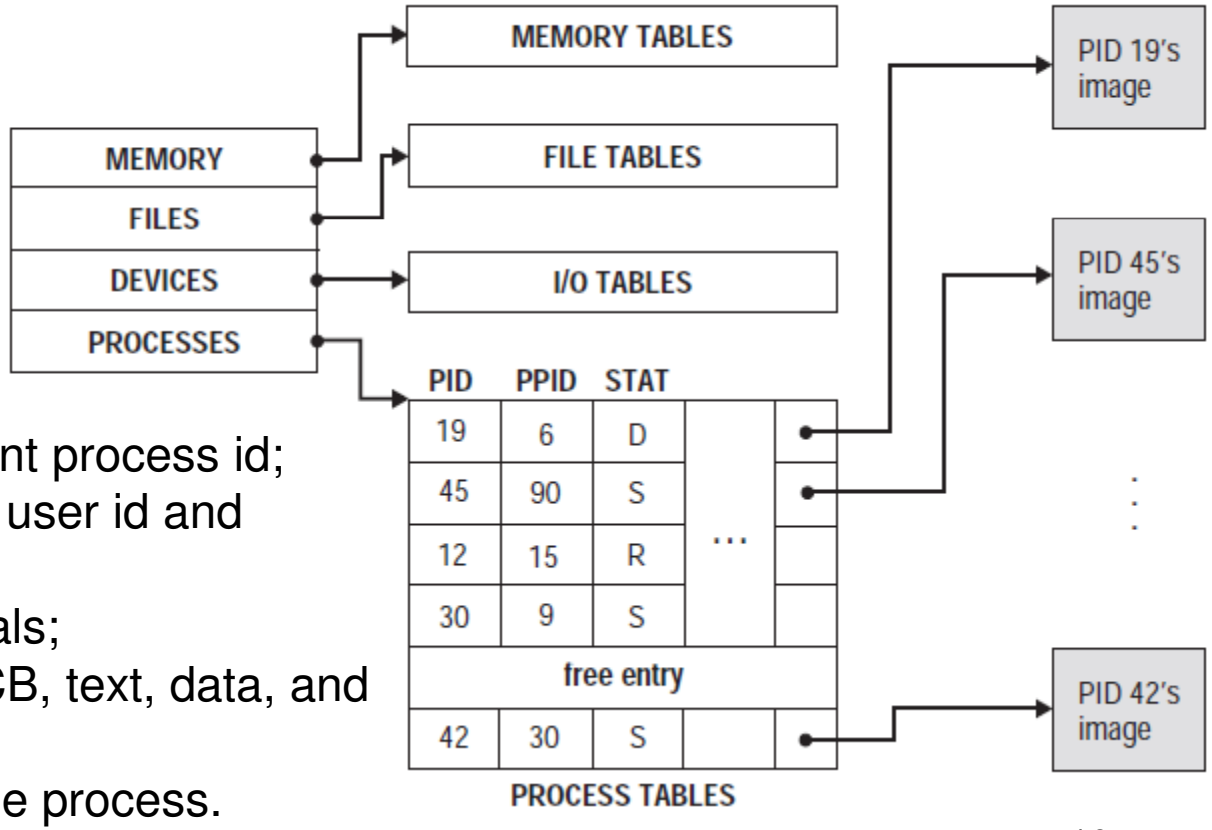


Process Table

Karoly.Bosa@jku.at

- The operating system has a table for all the resources of the computer that it including: processes, devices, memory, and files.

- The process table has an entry for each process image in memory.



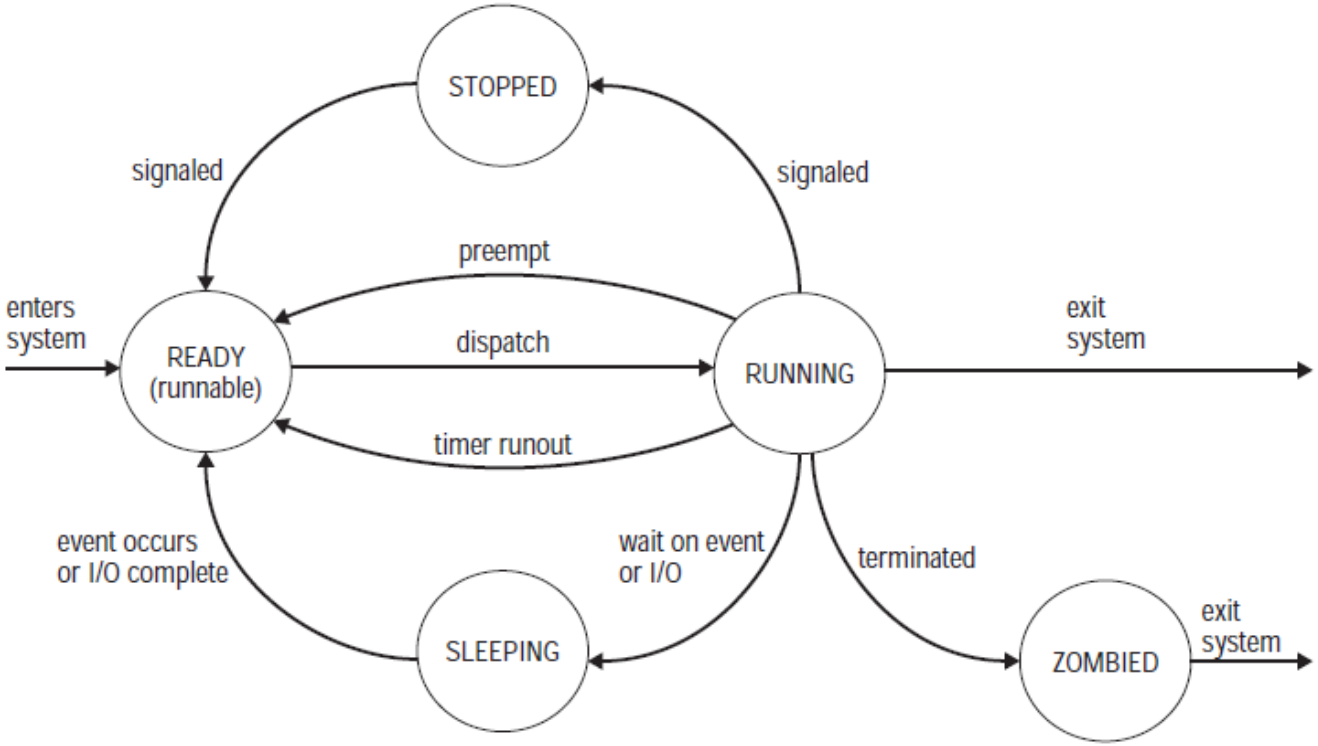
- Each entry contains:
  - the process and parent process id;
  - the real and effective user id and group id;
  - a list of pending signals;
  - the location of the PCB, text, data, and stack segments and;
  - the *current state* of the process.

Process States

- During a process's execution, it changes its state.
- The *state* of the process is the current condition or status of the process.
- In a POSIX - compliant environment, a process can be in the following states:
  - Running
  - Runnable (ready)
  - Zombied
  - Waiting (blocked)
  - Stopped
- *State transition* is the circumstance that causes the process to change its state.

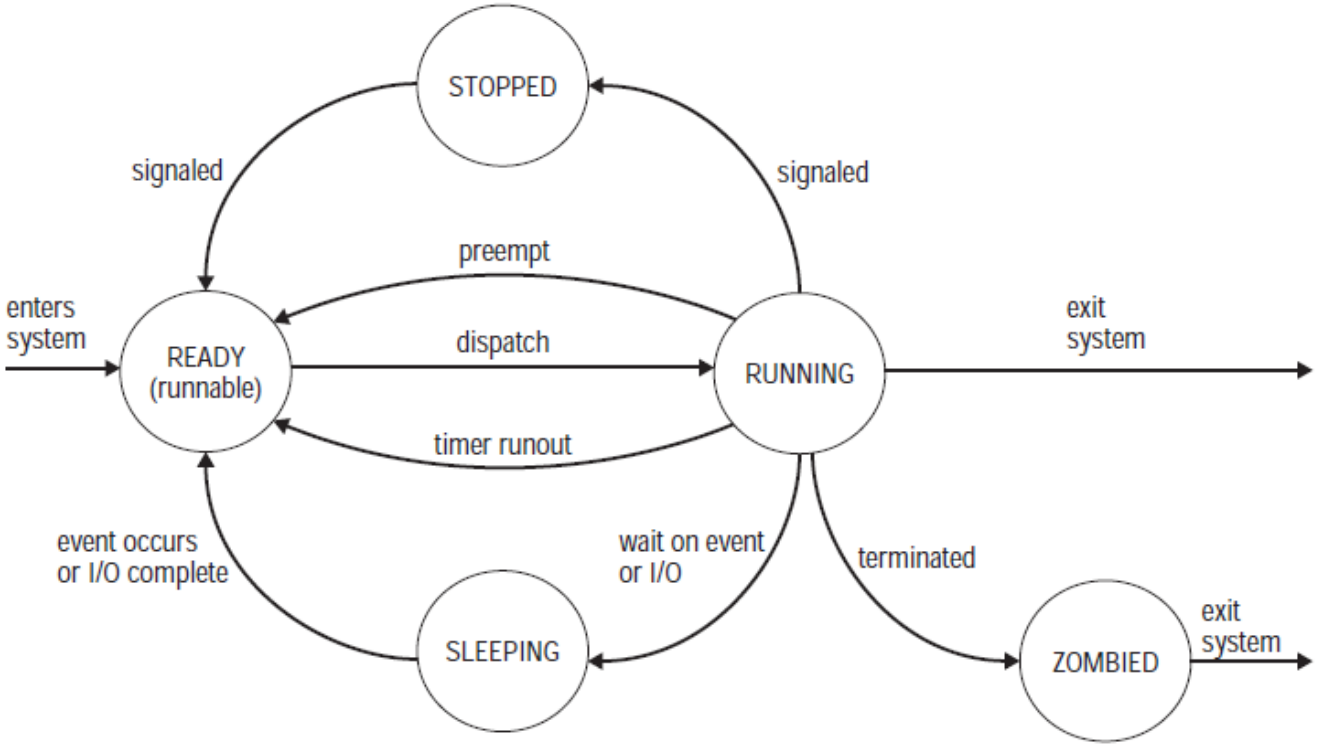
State Transitions I.

READY->RUNNING (dispatch)	The process is assigned to the processor.
RUNNING-> SLEEPING (block)	The process gives up the processor before the time slice has run out. The process may need to wait for an event or has made a system call, for example, a request for I/O. The process is placed in a queue with other sleeping processes.



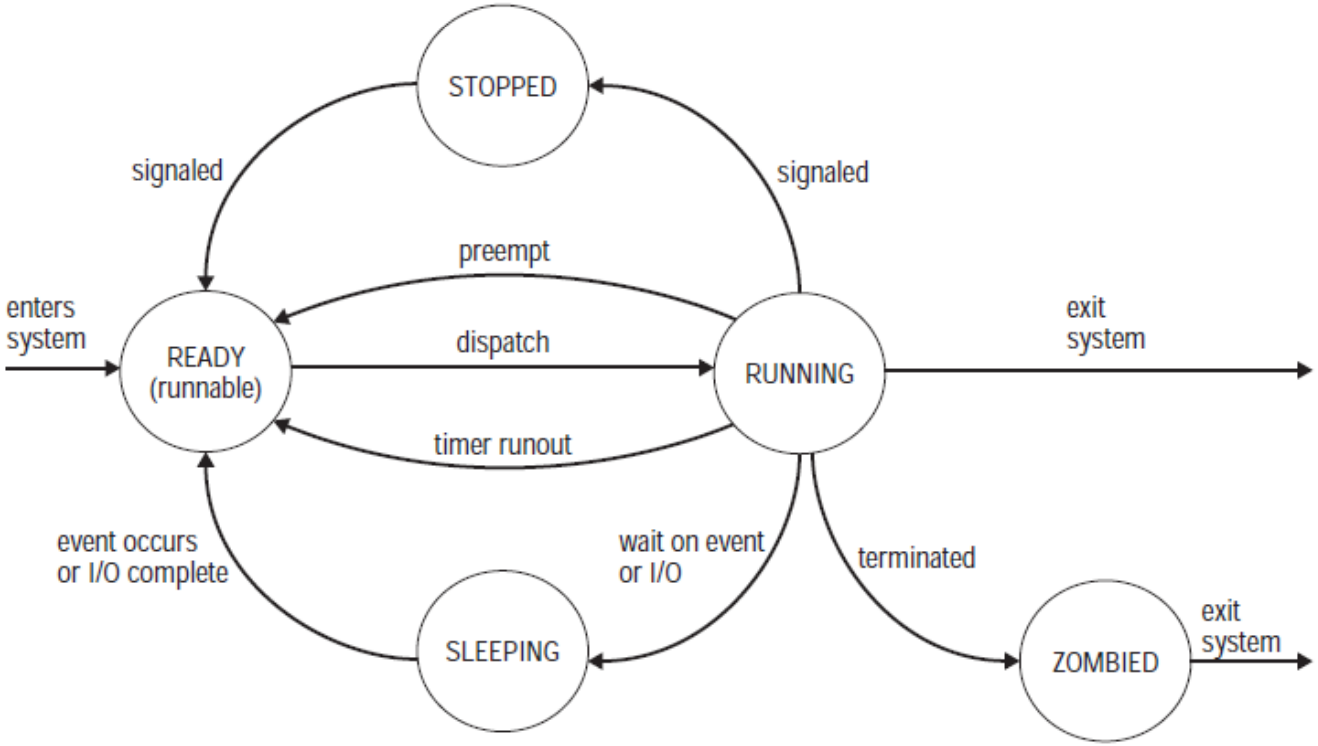
State Transitions II.

RUNNING-> READY(preempt)	The process has been preempted before the time slice ran out. This can occur if a process with a higher priority is runnable. The process is placed back in the ready queue.
RUNNING-> READY(timer runout)	The time slice the process assigned to the processor has run out. The process is placed back in the ready queue.



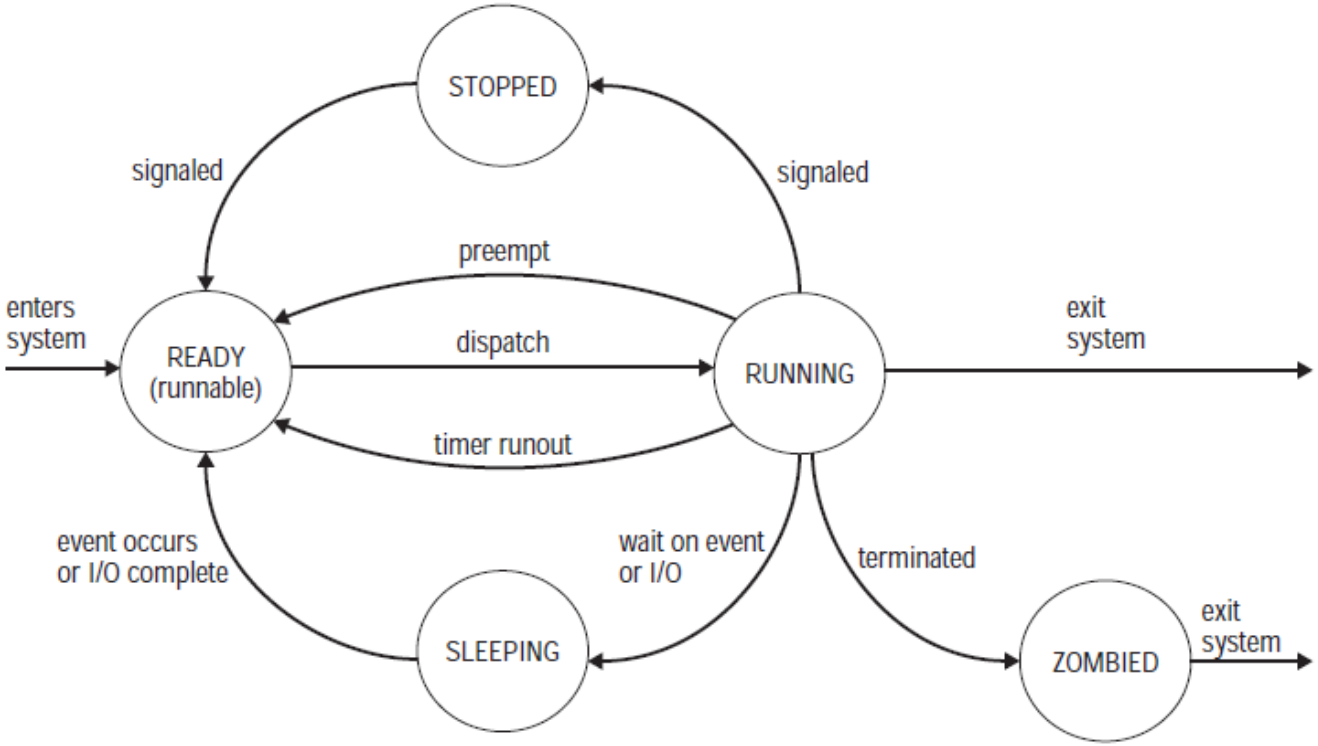
State Transitions III.

SLEEPING->READY (unblock)	The event the process was waiting for has occurred, or the system call has completed. For example, the I/O request is filled. The process is placed back in the ready queue.
RUNNING-> STOPPED	The process gives up the processor because it has received a signal to stop.



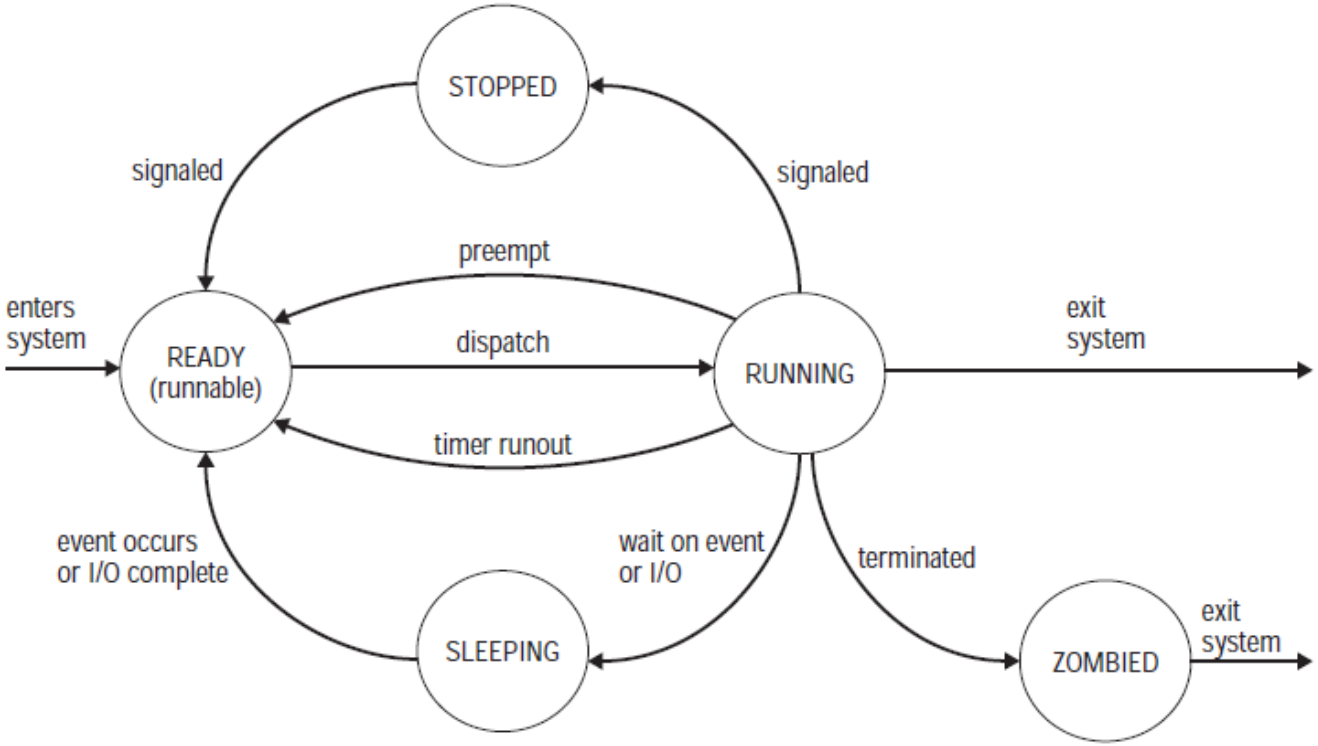
State Transitions IV.

STOPPED->READY	The process has received the signal to continue and is placed back in the ready queue.
RUNNING->ZOMBIED	The process has been terminated and awaits the parent to retrieve its exit status from the process table.



State Transitions V.

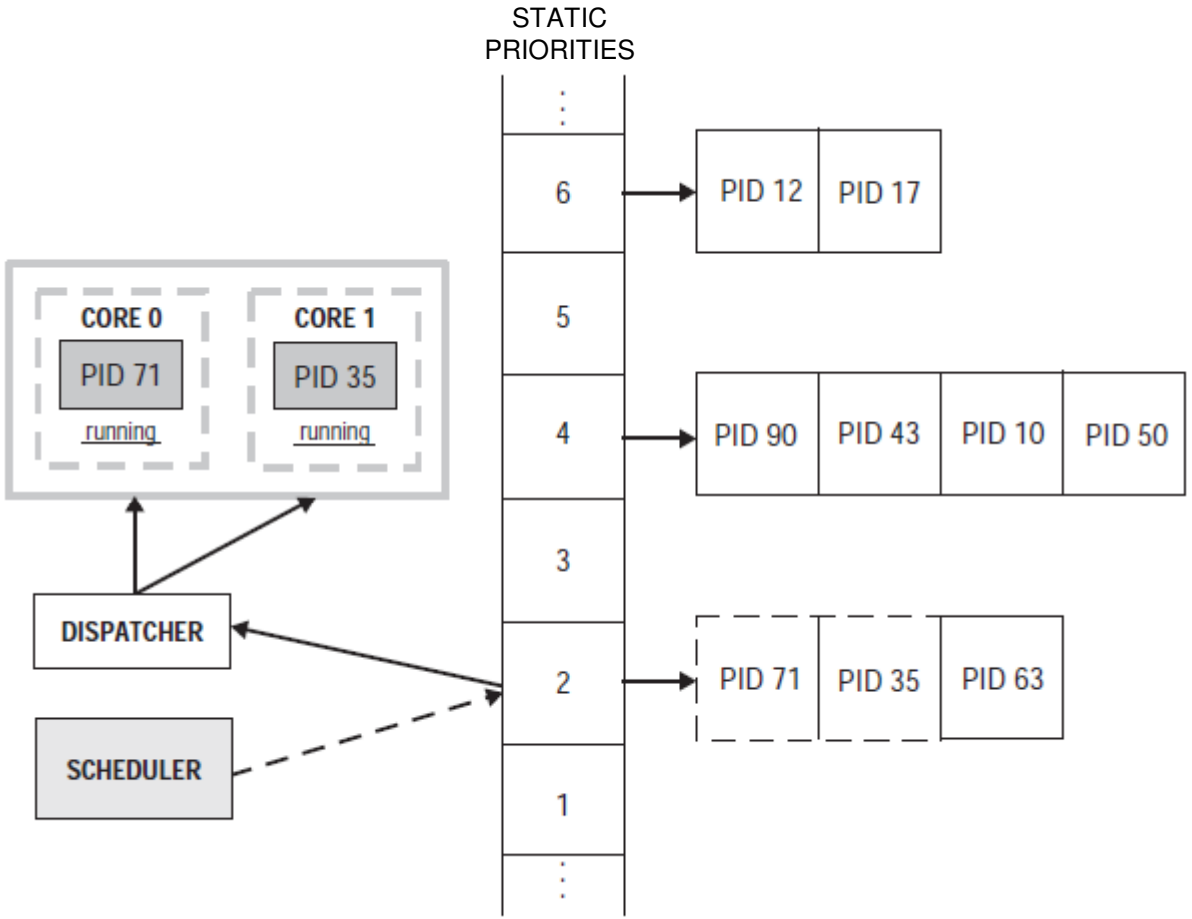
ZOMBIED->EXIT	The parent process has retrieved the exit status, and the process exits the system.
RUNNING->EXIT	The process has terminated, the parent has retrieved the exit status, and the process exits the system.



Process Scheduling: Multilevel priority Queue

Karoly.Bosa@jku.at

- Each process is given a priority class and placed in a priority queue with other runnable processes with the same priority class.
- The scheduler assigns the process at the head of the nonempty highest priority queue to the processor.

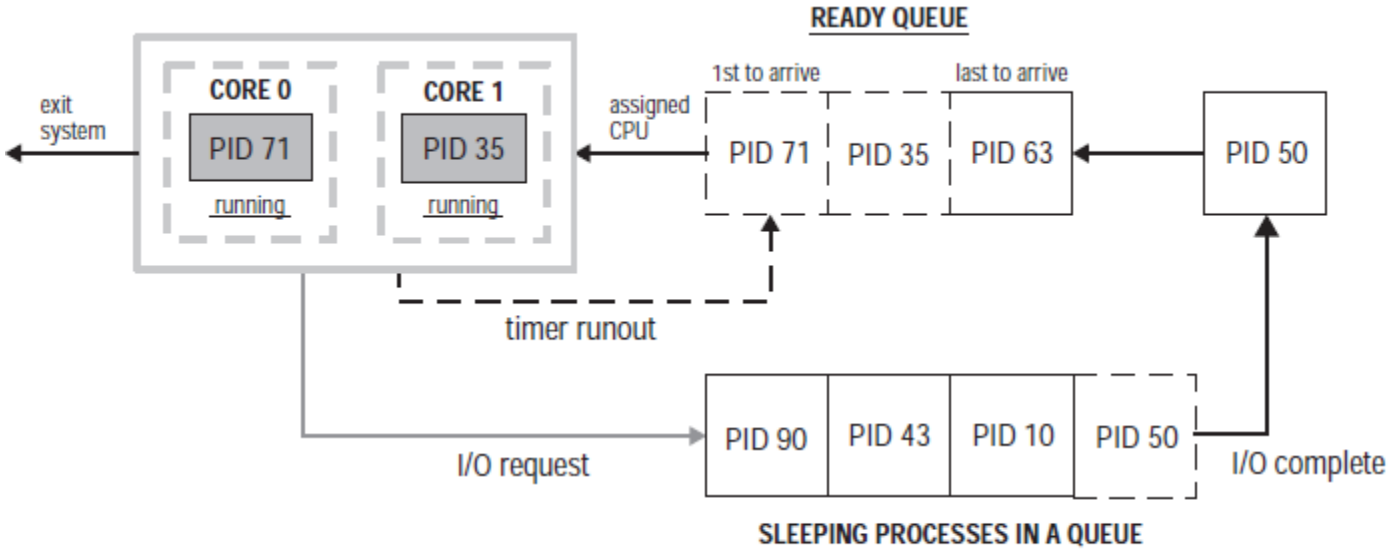


- Priorities can be ***dynamic*** or ***static*** in Linux/Unix*:*
  - Once a static priority of a process is set, it cannot be changed.
  - Dynamic priorities (nice value) can be changed.

- System processes have a higher priority than user processes.

Scheduling Policy: First In, First Out (FIFO)

Karoly.Bosa@jku.at

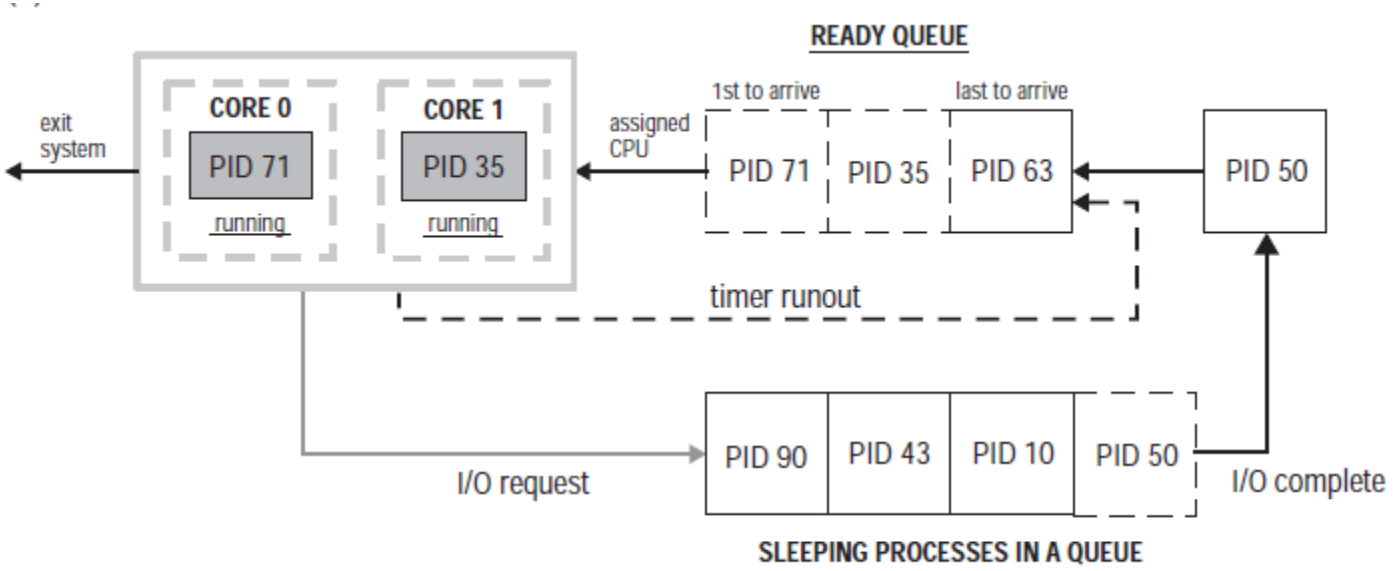


- With a FIFO scheduling policy, processes are assigned the processor according to the arrival time in the queue.

- A process can make a system call and give up the processor to another process with the same priority level. The process is then placed at the end of its priority queue.

Scheduling Policy: Round Robin (RR)

Karoly.Bosa@jku.at



- RR scheduling is the same as FIFO scheduling with one exception: When the time slice expires, the process is placed at the back of the queue and the next process in the queue is assigned the processor.

Scheduling Policy: OTHER

- It is defined as architecture dependent scheduling in POSIX. In Linux/Unix systems it works as follows:
  - The Linux kernel implements a dynamic priority ranges.
  - This is the nice value, a number from –20 to 19 with a default of zero.
  - This kind of priority applies only to conventional processes (with 0 static priority).
  - Processes with a lower nice value (higher priority) run before processes with a higher nice value (lower priority).
  - If a process is ready to run, but its execution is denied, its nice value is decremented in every quantum.

Context Switch

- A *context switch* occurs when the use of the processor is switched from one process to another process.

- A context switch occurs when a:
  - Process is preempted
  - Process voluntarily gives up the processor
  - Process makes an I/O request or needs to wait for an event
  - Process switches from user mode to kernel mode

- When a context switch occurs,
  - the system saves the context of the current running process and
  - Restores the context of the next process selected to use the processor.

Monitoring Processes with the ps Utility

Karoly.Bosa@jku.at

- The ps utility generates a report that summarizes execution statistics for the current processes.

Headers	Description	Headers	Description
USER, UID	Username of process owner	TT, TTY	Process's controlling terminal
PID PPID	Process ID Parent process ID	S, STAT	Current state of the process
PGID SID	ID of process group leader ID of session leader	TIME	Total CPU time used by the process (HH:MM:SS)
%CPU	Percentage of CPU time used by the process in the last minute	STIME, START	Time or date the process started
RSS	Amount of real RAM currently used by the process in k	NI	Nice value of the process
%MEM	Percentage of real RAM used by the process in the last minute	PRI	Priority of the process
SZ	Size of virtual memory of the process's data and stack in k or pages		
WCHAN	Address of an event for which a process is sleeping	ADDR	Memory address of a process
COMMAND CMD	Command name and arguments	LWP NLWP	ID of the lwp (thread) The number of lwps

Usage of ps Utility


```
(Linux)
ps -[Unix98 options]
   [BSD-style options]
   --[GNU-style long options

(Solaris)
ps [-aAdeflcjLPy][-o format][-t termlist][-u userlist]
   [-G grouplist][-p proclist][-g pgrplist][-s sidlist]
```

To see every process on the system:

–ps –e (using standard syntax)

–ps ax (using BSD syntax)

To print a process tree:

–ps –ejH

–ps axjf

Some options:

- **-t *term*** : List the processes associated with the terminal specified by term.
- **T :** (Linux) All processes in this terminal

Some options:

- **-u *username*** : List the processes belong to the given user
- **- e** : All current processes
- **r :** (Linux) Only running processes
- **- f** : Full listings
- **- l** : Long format
- **-o *format***: specify output format

Example of the Usage ps Utility

Karoly.Bosa@jku.at

```
$ ps -f
     UID   PID  PPID  C    STIME    TTY   TIME CMD
 cameron  2214  2212  0 21:03:35 pts/12   0:00 -ksh
 cameron  2396  2214  2 11:55:49 pts/12   0:01 nedit

$ ps -lf
F S     UID   PID  PPID  C PRI NI     ADDR  SZ    WCHAN    STIME    TTY TIME   CMD
8 S cameron  2214  2212  0  51 20 70e80f00 230 70e80f6c 21:03:35 pts/12 0:00  -ksh
8 S cameron  2396  2214  1  53 24 70d747b8 843 70152aba 11:55:49 pts/12 0:01 nedit

$ ps -lfP
F S UID         PID  PPID PSR  C PRI  NI ADDR SZ WCHAN  STIME TTY          TIME CMD
0 S kbosa      4318  4317   1  0  80   0 -  1157 -      19:37 pts/0    00:00:00 -tcsh
0 R kbosa      4543  4318   0  0  80   0 -   599 -      19:55 pts/0    00:00:00 ps -lfP

$ ps Tux
USER       PID %CPU %MEM    VSZ   RSS    TTY  STAT    START   TIME COMMAND
tdhughes 19259  0.0  0.1   2448 1356   pts/4     S    20:29   0:00 -bash
tdhughes 19334  0.0  0.0   1732  860   pts/4     S    20:33   0:00 /home/tdhughes/pv
tdhughes 19336  0.0  0.0   1928  780   pts/4     S    20:33   0:00 /home/tdhughes/pv
tdhughes 19337 18.0  2.4  26872 24856 pts/4     R    20:33   0:47 /home/tdhughes/pv
tdhughes 19338 18.0  2.3  26872 24696 pts/4     R    20:33   0:47 /home/tdhughes/pv
tdhughes 19341 17.9  2.3  26872 24556 pts/4     R    20:33   0:47 /home/tdhughes/pv
tdhughes 19400  0.0  0.0   2544  692   pts/4     R    20:38   0:00 ps Tux
tdhughes 19401  0.0  0.1   2448 1356   pts/4     R    20:38   0:00 -bash
```

Example of the Usage ps Utility

Karoly.Bosa@jku.at

```
$ ps -f
     UID   PID  PPID  C    STIME    TTY   TIME CMD
 cameron  2214  2212  0 21:03:35 pts/12   0:00 -ksh
 cameron  2396  2214  2 11:55:49 pts/12   0:01 nedit

$ ps -lf
 F S     UID   PID  PPID  C PRI NI     ADDR  SZ    WCHAN    STIME    TTY TIME   CMD
 8 S cameron  2214  2212  0  51 20 70e80f00 230 70e80f6c 21:03:35 pts/12 0:00  -ksh
 8 S cameron  2396  2214  1  53 24 70d747b8 843 70152aba 11:55:49 pts/12 0:01 nedit

$ ps -lfP
F S UID         PID  PPID PSR  C PRI  NI ADDR SZ WCHAN  STIME TTY          TIME CMD
0 S kbosa      4318  4317   1  0  80   0 -  1157 -      19:37 pts/0    00:00:00 -tcsh
0 R kbosa      4543  4318   0  0  80   0 -   599 -      19:55 pts/0    00:00:00 ps -lfP

$ ps Tux
USER       PID %CPU %MEM    VSZ   RSS    TTY  STAT    START   TIME COMMAND
tdhughes 19259  0.0  0.1   2448 1356   pts/4     S    20:29   0:00 -bash
tdhughes 19334  0.0  0.0   1732  860   pts/4     S    20:33   0:00 /home/tdhughes/pv
tdhughes 19336  0.0  0.0   1928  780   pts/4     S    20:33   0:00 /home/tdhughes/pv
tdhughes 19337 18.0  2.4  26872 24856 pts/4     R    20:33   0:47 /home/tdhughes/pv
tdhughes 19338 18.0  2.3  26872 24696 pts/4     R    20:33   0:47 /home/tdhughes/pv
tdhughes 19341 17.9  2.3  26872 24556 pts/4     R    20:33   0:47 /home/tdhughes/pv
tdhughes 19400  0.0  0.0   2544  692   pts/4     R    20:38   0:00 ps Tux
tdhughes 19401  0.0  0.1   2448 1356   pts/4     R    20:38   0:00 -bash
```

Example of the Usage ps Utility

Karoly.Bosa@jku.at

```
$ ps -f
     UID   PID  PPID  C    STIME    TTY   TIME CMD
 cameron  2214  2212  0 21:03:35 pts/12   0:00 -ksh
 cameron  2396  2214  2 11:55:49 pts/12   0:01 nedit

$ ps -lf
F S     UID   PID  PPID  C PRI NI     ADDR  SZ    WCHAN    STIME    TTY TIME   CMD
8 S cameron  2214  2212  0  51 20 70e80f00 230 70e80f6c 21:03:35 pts/12 0:00  -ksh
8 S cameron  2396  2214  1  53 24 70d747b8 843 70152aba 11:55:49 pts/12 0:01 nedit

$ ps -lfP
F S UID        PID  PPID PSR  C PRI  NI ADDR SZ WCHAN  STIME TTY          TIME CMD
0 S kbosa     4318  4317   1  0  80   0 -  1157 -      19:37 pts/0    00:00:00 -tcsh
0 R kbosa     4543  4318   0  0  80   0 -   599 -      19:55 pts/0    00:00:00 ps -lfP

$ ps Tux
USER       PID %CPU %MEM    VSZ   RSS    TTY  STAT    START   TIME COMMAND
tdhughes 19259  0.0  0.1   2448 1356   pts/4     S    20:29   0:00 -bash
tdhughes 19334  0.0  0.0   1732  860   pts/4     S    20:33   0:00 /home/tdhughes/pv
tdhughes 19336  0.0  0.0   1928  780   pts/4     S    20:33   0:00 /home/tdhughes/pv
tdhughes 19337 18.0  2.4  26872 24856 pts/4     R    20:33   0:47 /home/tdhughes/pv
tdhughes 19338 18.0  2.3  26872 24696 pts/4     R    20:33   0:47 /home/tdhughes/pv
tdhughes 19341 17.9  2.3  26872 24556 pts/4     R    20:33   0:47 /home/tdhughes/pv
tdhughes 19400  0.0  0.0   2544  692   pts/4     R    20:38   0:00 ps Tux
tdhughes 19401  0.0  0.1   2448 1356   pts/4     R    20:38   0:00 -bash
```

Example of the Usage ps Utility

Karoly.Bosa@jku.at

```
$ ps -f
     UID   PID  PPID  C    STIME    TTY   TIME CMD
 cameron  2214  2212  0 21:03:35 pts/12   0:00 -ksh
 cameron  2396  2214  2 11:55:49 pts/12   0:01 nedit

$ ps -lf
F S     UID   PID  PPID  C PRI NI     ADDR  SZ    WCHAN    STIME    TTY TIME   CMD
8 S cameron  2214  2212  0  51 20 70e80f00 230 70e80f6c 21:03:35 pts/12 0:00  -ksh
8 S cameron  2396  2214  1  53 24 70d747b8 843 70152aba 11:55:49 pts/12 0:01 nedit

$ ps -lfP
F S UID         PID  PPID PSR  C PRI  NI ADDR SZ WCHAN  STIME TTY          TIME CMD
0 S kbosa      4318  4317   1  0  80   0 -  1157 -      19:37 pts/0    00:00:00 -tcsh
0 R kbosa      4543  4318   0  0  80   0 -   599 -      19:55 pts/0    00:00:00 ps -lfP

$ ps Tux
USER       PID %CPU %MEM   VSZ  RSS   TTY  STAT    START   TIME COMMAND
tdhughes 19259  0.0  0.1  2448 1356  pts/4    S    20:29   0:00 -bash
tdhughes 19334  0.0  0.0  1732  860  pts/4    S    20:33   0:00 /home/tdhughes/pv
tdhughes 19336  0.0  0.0  1928  780  pts/4    S    20:33   0:00 /home/tdhughes/pv
tdhughes 19337 18.0  2.4 26872 24856 pts/4    R    20:33   0:47 /home/tdhughes/pv
tdhughes 19338 18.0  2.3 26872 24696 pts/4    R    20:33   0:47 /home/tdhughes/pv
tdhughes 19341 17.9  2.3 26872 24556 pts/4    R    20:33   0:47 /home/tdhughes/pv
tdhughes 19400  0.0  0.0  2544  692  pts/4    R    20:38   0:00 ps Tux
tdhughes 19401  0.0  0.1  2448 1356  pts/4    R    20:38   0:00 -bash
```

Example of the Usage ps Utility

Karoly.Bosa@jku.at

```
$ ps Tux
USER        PID %CPU %MEM    VSZ   RSS    TTY   STAT    START   TIME COMMAND
tdhughes  19259  0.0  0.1   2448  1356   pts/4     S    20:29   0:00 -bash
tdhughes  19334  0.0  0.0   1732   860   pts/4     S    20:33   0:00 /home/tdhughes/pv
tdhughes  19336  0.0  0.0   1928   780   pts/4     S    20:33   0:00 /home/tdhughes/pv
tdhughes  19337 18.0  2.4  26872 24856  pts/4     R    20:33   0:47 /home/tdhughes/pv
tdhughes  19338 18.0  2.3  26872 24696  pts/4     R    20:33   0:47 /home/tdhughes/pv
tdhughes  19341 17.9  2.3  26872 24556  pts/4     R    20:33   0:47 /home/tdhughes/pv
tdhughes  19400  0.0  0.0   2544   692   pts/4     R    20:38   0:00 ps Tux
tdhughes  19401  0.0  0.1   2448  1356   pts/4     R    20:38   0:00 -bash
```

Status of Process	Description
D	Uninterruptible sleep (usually I/O)
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
Z	"Zombie" process, terminated with no parent

The STAT header can reveal additional information about the status of the process:

- D: (BSD) Disk wait
- P: (BSD) Page wait
- X: (System V) Growing: waiting for memory
- W: (BSD) Swapped out
- K: (AIX) Available kernel process
- N: (BSD) Niced: execution priority lowered
- >: (BSD) Niced: execution priority artificially raised
- <: (Linux) High-priority process
- L: (Linux) Pages are locked in memory

Creating Processes

- fork() - function creates a new process. The new process (child process) shall be an exact copy of the calling process (parent process) except as some minor things (e.g.:process id).

- exec() - The *exec* family of functions replaces the current process image with a new process image.

- system() – executes a command specified in the argument string and returns after the command has been completed.

- posix_spawn() - create child processes with more fine-grained control during creation…

Remark: These functions also control the attributes that the child process inherits from the parent process, e.g.:

File descriptors,	scheduling policy,	process group id,
user and group id,	signal mask.	

Using the fork() Function Call


```
#include <unistd.h>

pid_t fork(void);
```

- It creates a new process that is a duplication of the calling process, the parent.
- The fork() returns two values if it succeeds:
    - It returns 0 to the child process and
    - it returns the PID of the child to the parent process.
- Both processes continue to execute from the instruction immediately following the fork() call.
- The fork() fails if the system does not have the resources to create another process.

Using the exec() Family of System Calls

- The exec family of functions replaces the calling process image with a new process image.

```
#include <unistd.h>

int execl(const char *path,const char *arg0,.../*,(char *)0 */);
int execle(const char *path,const char *arg0,.../*,
          (char *)0 *,char *const envp[]*/);
int execv(const char *path,char *const arg[]);
int execlp(const char *file,const char *arg0,.../*,(char *)0 */);
int execve(const char *path,char *const arg[],
           char *const envp[]);
int execvp(const char *file,char *const arg[]);
```

- The new process image is a regular executable file:
  - The executable can be specified as a path or a filename.
  - These functions can pass command-line arguments to the new process.
  - Environment variables can also be specified.

- Usually there is no return value if the function is not successful, because the process image that contained the call to the exec is overwritten.

- All of the exec() functions can fail under these conditions:
  - Permissions are denied.
  - Files do not exist.
  - File is not executable
  - Problems with symbolic links

Example for fork() and execl() Calls

- The exec functions are often used together with the fork()

```
#include <unistd.h>
//...
RetVal = fork();
If (RetVal == 0) {
        execl("/bin/ls", "ls", "-l", NULL);
}
//…
```

- Remark: If RetVal is 0, then it is the child process.

Execl Functions

- The **execl() , execle()** , and **execlp()** functions pass the command-line arguments as a list.

- The number of command - line arguments should be known at compilation time

- **int execl(const char *path, const char *arg0,.../*,(char *)0 */);**

- **int execle(const char *path, const char *arg0,.../*,(char *)0 *, char *const envp[]*/);**

  envp[] parameter contains the new environment for the new process. It is a pointer to a null–terminated array of null-terminated strings. Each string has the form: name=value , e.g.:

  char *const envp[] = {“PATH=/opt/kde5:/sbin”, “HOME=/home”,NULL};

- **int execlp(const char *file, const char *arg0,.../*,(char *)0 */);**

  file is the name of the program executable. It uses the PATH environment variable to locate the executables.

Execv Functions

- The **execv() , execve()** , and **execvp()** functions pass the command-line arguments in a vector of pointers to null-terminated strings.
- The number of command-line arguments should be known at compilation time.
- **int execv(const char *path,char *const arg[]);**
- **int execve(const char *path,char *const arg[],char *const envp[]);**

  This function is identical to execv() except that it has the additional parameter envp[] described earlier.
- **int execvp(const char *file,char *const arg[]);**

  file is the name of the program executable.

Examples for Exec Functions


```
char *const envp[] = {“HOME=/home”,NULL};

execl(“/bin/ls”, ”ls”, ”-l”, NULL);
execle(“/bin/ls”, ”ls”, ”-l”, “$HOME”, NULL, envp);
execlp(“ls”, ”ls”, ”-l”, NULL);
```

```
char *const arg[] = {“du”, “-h”, ”/etc”,  “/root”, NULL};
char *const envp[] = {“HOME=/home”, NULL};

execv(“/usr/bin/du”, arg);
execve(“/usr/bin/du”, arg, envp);
execvp(“du”, arg);
```

Determining the Restrictions of exec() Functions

- There is a limit on the size that argv[] and envp[] can be when passed to the exec() functions.
- The sysconf() can be used to determine these limitations.
- To determine the maximum size of the command-line arguments plus the size of environment variables for the functions:

```
#include < unistd.h >
//…
sysconf (_SC_ARG_MAX);
```

- To determine the maximum number of simultaneous processes allowed per user id:

```
sysconf (_SC_CHILD_MAX);
```

Using system() Functions to Spawn Processes

- The **system()** function executes a fork() , and the child process calls an exec() with a shell that executes the given command or program.

```
#include <stdlib.h>

int system(const char *string);
```

- The string parameter can be a system command or the name of an executable file.

- Errors can happen at several levels:
  - The function returns the value127 if the exec() fails,
  - - 1 if some other error occurs and
  - the return code of the command is returned if the function succeeds.

The function posix_spawn()


```
#include <spawn.h>

int posix_spawn(pid_t *restrict pid, const char *restrict path,
                const posix_spawn_file_actions_t *file_actions,
                const posix_spawnattr_t *restrict attrp,
                char *const argv[restrict],
                char *const envp[restrict]);
int posix_spawnp(pid_t *restrict pid, const char *restrict file,
                 const posix_spawn_file_actions_t *file_actions,
                 const posix_spawnattr_t *restrict attrp,
                 char *const argv[restrict],
                 char *const envp[restrict]);
```

Remark: POSIX functions used to spawn and manage processes are contained by the header spawn.h.

The file_actions Parameter

- The file_actions parameter is a pointer to a posix_spawn_file_actions_t structure:

```
struct posix_spawn_file_actions_t{
        {
           int __allocated;
           int __used;
           struct __spawn_action *actions;
           int __pad[16];
        };
```

- This data structure that contains information about the actions to be performed in the new process with respect to file descriptors.

File Action Attribute Functions I.

Karoly.Bosa@jku.at

int posix_spawn_file_actions_addclose (posix_spawn_file_actions_t *file_actions, int fildes);	Adds a close() action to a spawn file action object specified by file_actions. This causes the file descriptor *fildes* to be closed when the new process is spawned using this file action object.
int posix_spawn_file_actions_addopen (posix_spawn_file_actions_t *file_actions, int fildes, const char *restrict path, int oflag, mode_t mode);	Adds an open() action to a spawn file action object specified by file_actions. This causes the file named path with the returned file descriptor *fildes* to be opened when the new process is spawned using this file action object.
int posix_spawn_file_actions_adddup2 (posix_spawn_file_actions_t *file_actions, int fildes, int newfildes);	Adds a dup2() action to a spawn file action object specified by file_actions. This causes the file descriptor *fildes* to be duplicated with the file descriptor *newfildes* when the new process is spawned using this file action object.

File Action Attribute Functions II.

int posix_spawn_file_actions_destroy (posix_spawn_file_actions_t *file_actions);	Destroys the specified file_actions object. This causes the object to be uninitialized. The object can then be reinitialized using posix_spawn_file_actions_init().
int posix_spawn_file_actions_init (posix_spawn_file_actions_t *file_actions);	Initializes the specified file_actions object. Once initialized, it contains no file actions to be performed.

The attrp Parameter

- The attrp parameter points to a posix_spawnattr_t structure:

```
struct posix_spawnattr_t
{
        short int __flags;
        pid_t __pgrp;
        sigset_t __sd;
        sigset_t __ss;
        struct sched_param __sp;
        int __policy;
        int __pad[16];
}
```

- This structure contains information about the scheduling policy, process group, signals, and flags for the new process.

The attributes of the posix_spawnattr_t Structure

- **__flags** : Used to indicate which process attributes are to be modified in the spawned process. They are bitwise - inclusive OR of 0 or more of the following:
  - POSIX_SPAWN_RESETIDS
  - POSIX_SPAWN_SETPGROUP
  - POSIX_SPAWN_SETSIGDEF
  - POSIX_SPAWN_SETSIGMASK
  - POSIX_SPAWN_SETSCHEDPARAM
  - POSIX_SPAWN_SETSCHEDULER
- **__pgrp** : The id of the process group to be joined by the new process.
- **__sd** : Represents the set of signals to be forced to use default signal handling by the new process.
- **__ss** : Represents the signal mask to be used by the new process.
- **__sp** : Represents the scheduling parameter to be assigned to the new process.
- **__policy** : Represents the scheduling policy to be used by the new process.

Spawn Process Attributes Functions I.

Karoly.Bosa@jku.at

int posix_spawnattr_getflags (const posix_spawnattr_t *restrict attr, short *restrict flags);	Returns the value of the __flags attribute stored in the specified attr object.
int posix_spawnattr_setflags (posix_spawnattr_t *attr, short flags);	Sets the value of the __flags attribute stored in the specified attr object to flags.
int posix_spawnattr_getpgroup (const posix_spawnattr_t *restrict attr, pid_t *restrict pgroup);	Returns the value of the __pgroup attribute stored in the specified attr object and stores it in pgroup.

Spawn Process Attributes Functions II.

Karoly.Bosa@jku.at

int posix_spawnattr_setpgroup (posix_spawnattr_t *attr, pid_t pgroup);	Sets the value of the __pgroup attribute stored in the specified attr object to pgroup if POSIX_ SPAWN_SETPGROUP is set in the __flags attribute.
int posix_spawnattr_getschedparam (const posix_spawnattr_t *restrict attr, struct sched_param *restrict schedparam);	Returns the value of the __sp attribute stored in the specified attr object and stores it in schedparam.
int posix_spawnattr_setschedparam (posix_spawnattr_t *attr, const struct sched_param *restrict schedparam);	Sets the value of the __sp attribute stored in the specified attr object to schedparam if POSIX_SPAWN_ SETSCHEDPARAM is set in the __flags attribute.

Spawn Process Attributes Functions III.

Karoly.Bosa@jku.at

int posix_spawnattr_getschedpolicy (const posix_spawnattr_t *restrict attr, int *restrict schedpolicy);	Returns the value of the __policy attribute stored in the specified attr object and stores it in schedpolicy.
int posix_spawnattr_setschedpolicy (posix_spawnattr_t *attr, int schedpolicy);	Sets the value of the __policy attribute stored in the specified attr object to schedpolicy if POSIX_SPAWN_ SETSCHEDULER is set in the __flags attribute.
int posix_spawnattr_getsigdefault (const posix_spawnattr_t *restrict attr, sigset_t *restrict sigdefault);	Returns the value of the __sd attribute stored in the specified attr object and stores it in sigdefault.

Spawn Process Attributes Functions IV.

Karoly.Bosa@jku.at

int posix_spawnattr_setsigdefault (posix_spawnattr_t *attr, const sigset_t *restrict sigdefault);	Sets the value of the __sd attribute stored in the specified attr object to sigdefault if POSIX_SPAWN_ SETSIGDEF is set in the __flags attribute.
int posix_spawnattr_getsigmask (const posix_spawnattr_t *restrict attr, sigset_t *restrict sigmask);	Returns the value of the __ss attribute stored in the specified attr object and stores it in sigmask.
int posix_spawnattr_setsigmask (posix_spawnattr_t *restrict attr, const sigset_t *restrict sigmask);	Sets the value of the __ss attribute stored in the specified attr object to sigmask if POSIX_SPAWN_ SETSIGMASK is set in the __flags attribute.

Spawn Process Attributes Functions V.

Karoly.Bosa@jku.at

int posix_spawnattr_destroy (posix_spawnattr_t *attr);	Destroys the specified attr object. The object can then become reinitialized using posix_spawnattr_init().
int posix_spawnattr_init (posix_spawnattr_t *attr);	Initializes the specified attr object with default values for all of the attributes contained in the structure.

A Simple posix_spawn() Example


```
#include <spawn.h>
#include <stdio.h>
#include <errno.h>
#include <iostream>
{
   //...
   posix_spawnattr_t X;
   posix_spawn_file_actions_t Y;
   pid_t Pid;
   char * argv[] = {"/bin/ps","-lf",NULL};
   char * envp[] = {"PROCESSES=2"};
   posix_spawnattr_init(&X);
   posix_spawn_file_actions_init(&Y);
   posix_spawn(&Pid,"/bin/ps",&Y,&X,argv,envp);
   perror("posix_spawn");
   cout << "spawned PID: " << Pid << endl;
   //...
   return(0);

}
```

Who Is the Parent? Who Is the Child?

- There are two functions that return the process id (PID) of the process and parent process:

```
#include <unistd.h>

pid_t getpid(void);
pid_t getppid(void);
```

  - getpid() returns the process id of the calling process.
  - getppid() returns the parent id of the calling process.
- These functions are always successful; therefore no errors are defined.