# Major Requirements of an Operating System

- Interleave the execution of the number of processes to maximize processor utilization while providing reasonable response time
- Allocate resources to processes
- Support interprocess communication and user creation of processes

# The Process (abstraction)

- Also called a *task*
- Execution of an individual program
  - an executable program
  - associated data
  - execution context
- Can be traced
  - list the sequence of instructions that execute

# The Process

- In UNIX
  - Process is an instance of a running program.
  - Lifetime: fork/vfork->exec->exit
  - Well-defined hierarchy: parent,child,init,
  - System processes:
    - *init* process: the top process
    - *swapper* & *pagedeamon*
  - Orphans
    - the parent process is terminated.

```
[giorgio@gastone MasterINFN]$ cat p.c
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main()
{
    int pid, ppid, pid1;
    printf("My ID - %d\n",getpid());
    printf("My Parent ID - %d\n",getppid());
    return 0;
}
[giorgio@gastone MasterINFN]$ cc p.c
[giorgio@gastone MasterINFN]$ ps
  PID TTY          TIME CMD
 8582 pts/2    00:00:00 bash
18230 pts/2    00:00:00 ps
[giorgio@gastone MasterINFN]$ a.out
My ID - 18231
My Parent ID - 8582
[giorgio@gastone MasterINFN]$
```

# Dispatcher

- The program that moves the processor from one process to another
- Prevents a single process from monopolizing processor time
- It cannot just select the process that has been in the queue the longest because it may be blocked
  - Not-running
    - ready to execute
  - Blocked
    - waiting for I/O

# Process Creation

- Submission of a batch job
- User logs on
- Create to provide a service such as printing
- Spawned by an existing process

# Process Termination

- When:
  - batch job issues *Halt* instruction
  - User logs off
  - Process executes a service request to terminate
  - On *error* and *fault* conditions

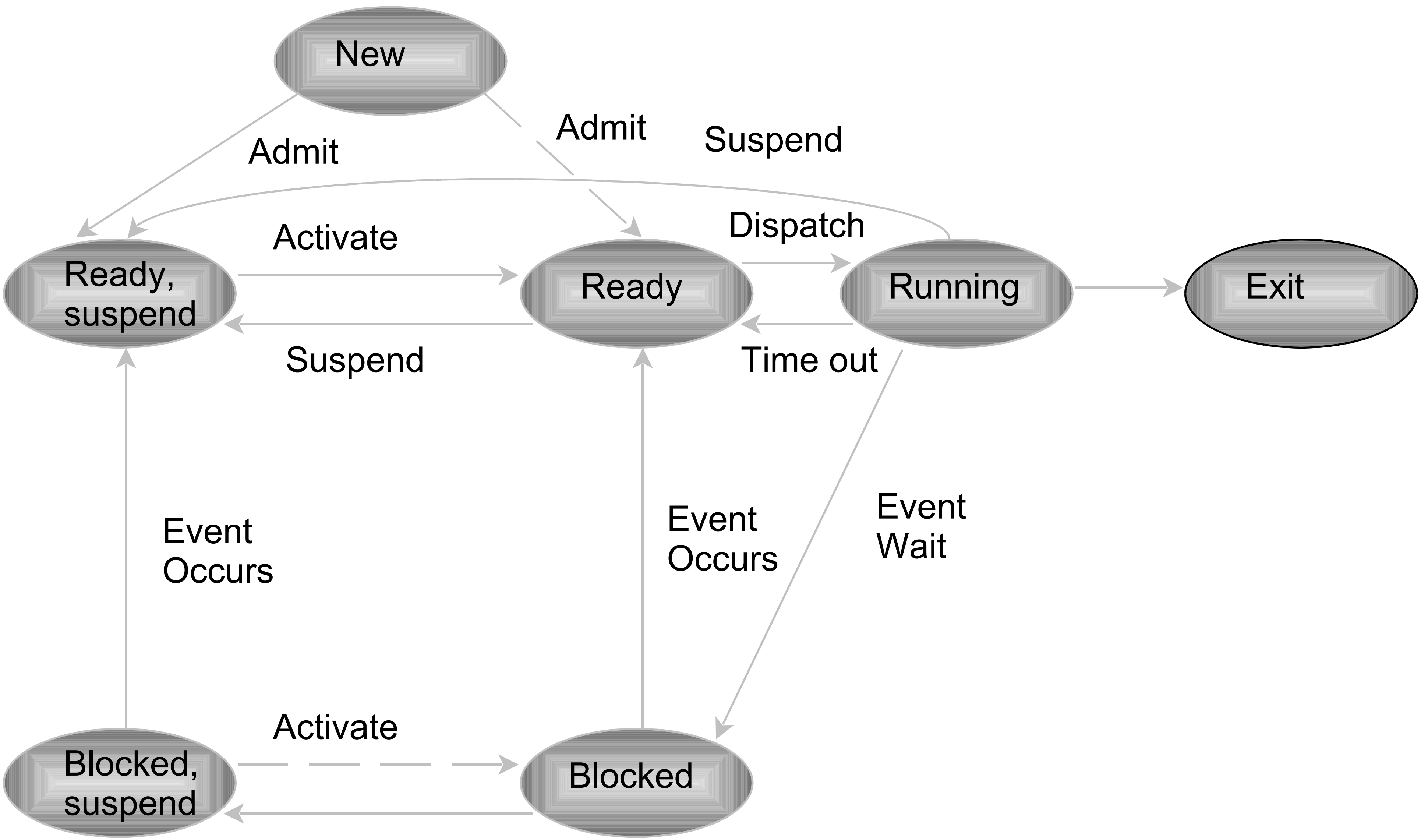
# Reasons for Process Termination

- Normal completion
- Time limit exceeded
- Memory unavailable
- Bounds violation
- Protection error
  - example write to read-only file
- Arithmetic error
- Time overrun
  - process waited longer than a specified maximum for an event

# Reasons for Process Termination

- I/O failure
- Invalid instruction
  - happens when try to execute data
- Privileged instruction
- Data misuse
- Operating system intervention
  - such as when deadlock occurs
- Parent terminates so child processes terminate
- Parent request

# Process State Transition Diagram with Two Suspend States



# Process Creation

- Assign a unique process identifier
- Allocate space for the process
- Initialize process control block
- Set up appropriate linkages
  - Ex: add new process to linked list used for scheduling queue
- Other
  - maintain an accounting file

# When to Switch a Process

- Interrupts
  - Clock
    - process has executed for the maximum allowable time slice
  - I/O
- Memory fault
  - memory address is in virtual memory so it must be brought into main memory
- Trap
  - error occurred
  - may cause process to be moved to *Exit* state
- Supervisor call
  - such as file open

# UNIX Process State

- Initial (idle)
- Ready to run
- Kernel/User running
- Zombie
- Asleep
- + (4BSD): stopped/suspend

# Process states and state transitions

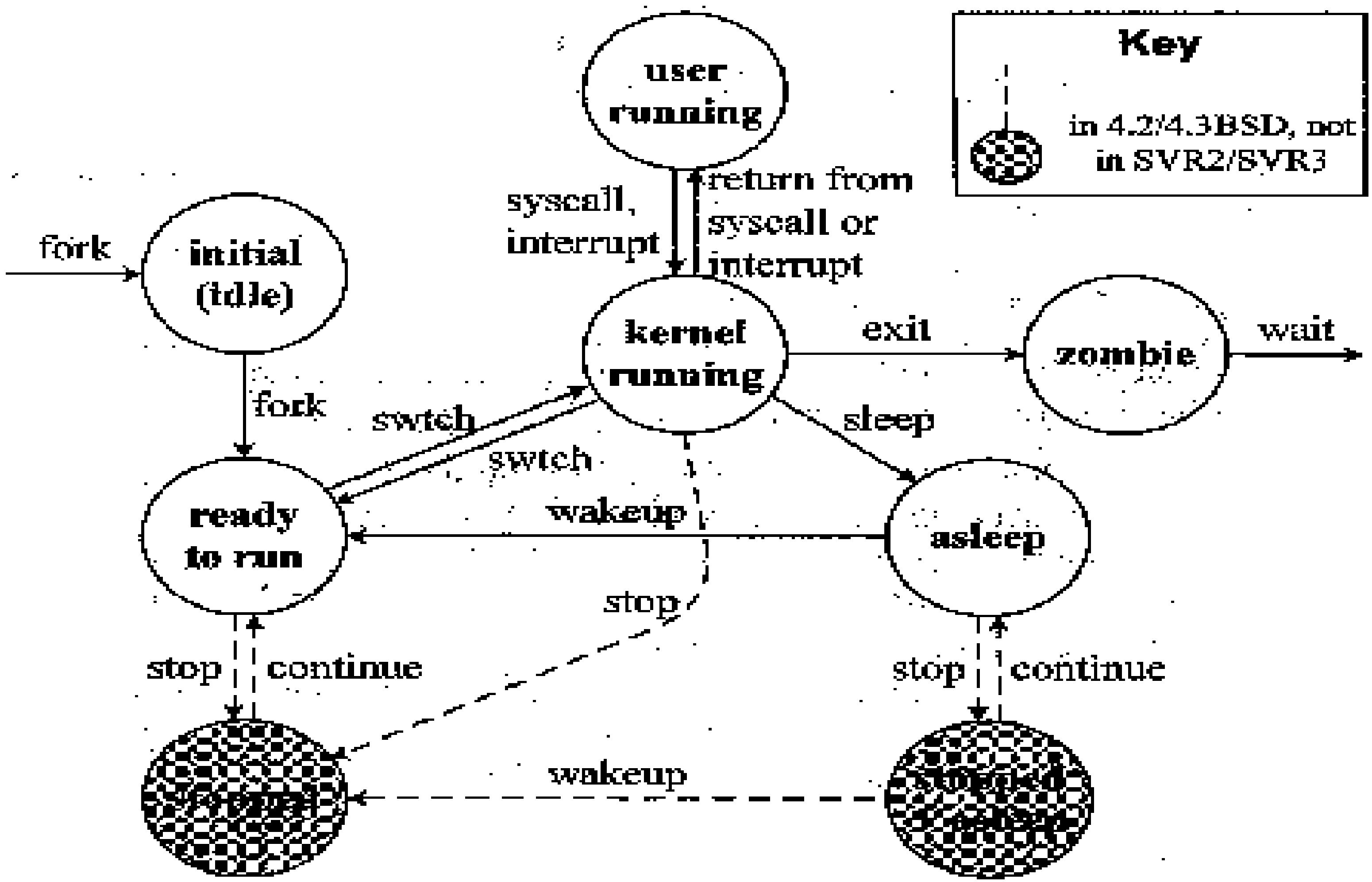



**Figure 2-3.** Process states and state transitions.

# Process Context

- User address space:
  - code, data, stack, shared memory regions
- Control information:
  - *u* area, proc, kernel stack, Addr.Trans. Map
- Credentials: *UID & GID*
- Environment variables:
  - inherited from the parent
- Hardware context(in PCB of u area):
  - PC, SP, PSW, MMR, FPU

# User Credentials

- Superuser: UID=0, GID=1
- Real IDs: login, send signals
- Effective IDs: file creation and access
- exec:
  - suid/sgid mode: set to that of the owner of the file
- setuid / setgid:

  SV & BSD are different with these

  - saved UID, saved GID in SV
  - setgroups() in BSD

# Who's who

- int getuid();
  - returns user id
- int getgid()
  - returns group id
- int geteuid();
  - return *effective* user id
- int getegid();
  - returns *effective* group id

# A typical process hierarchy in 4.3BSD UNIX

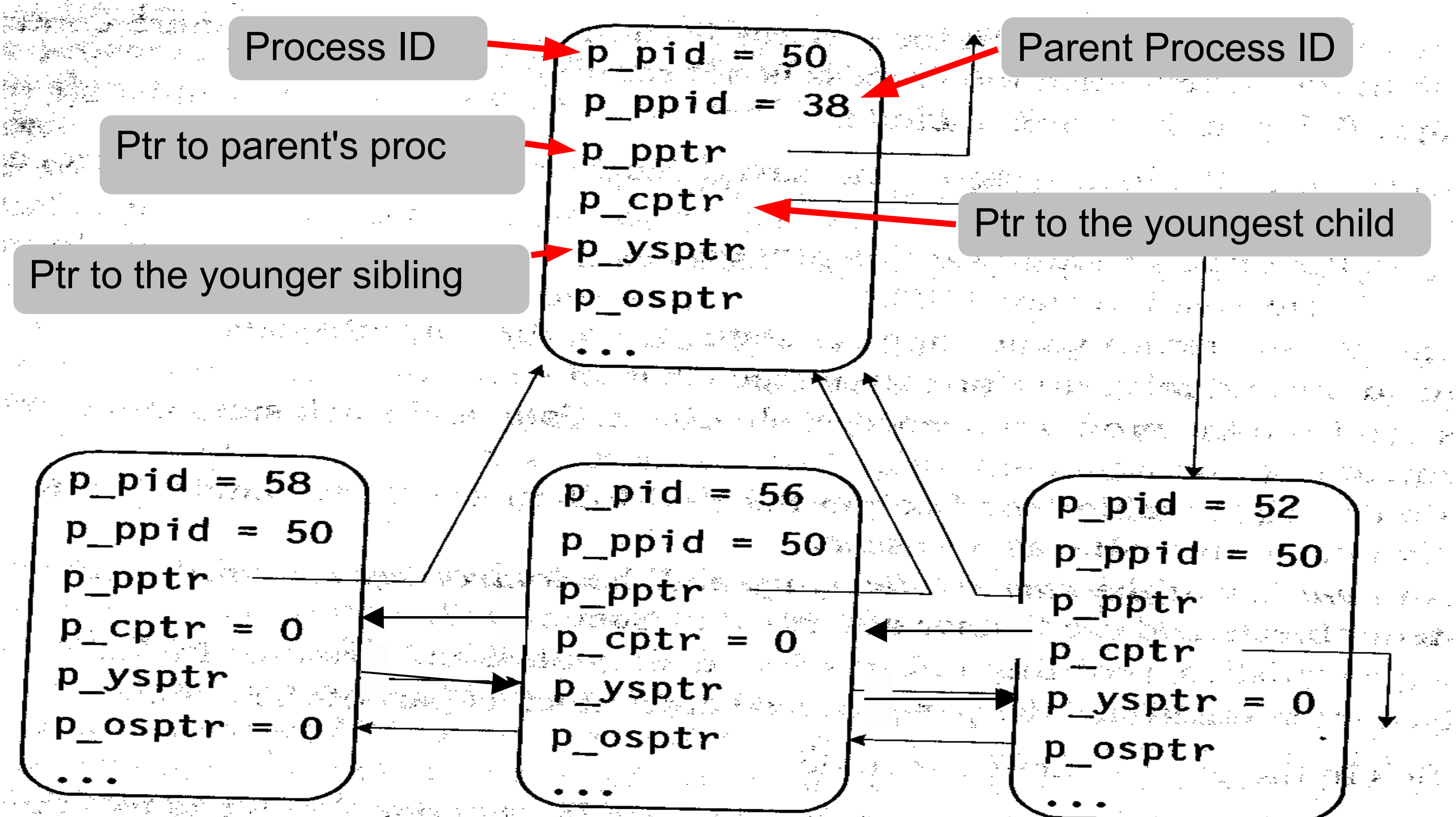



**Figure 2-4.** A typical process hierarchy in 4.3BSD UNIX.

# The UNIX kernel

- A special program that runs directly on the hardware.
- Implements the process model and services.
- Resides on disk
  - */vmunix, /unix, /vmlinuz, …*
- Bootstrapping: loads the kernel.
- Initializes the system and sets up the environment, remains in memory before shut down

# UNIX Services

- System Calls
- Hardware exceptions
  - Divide by 0, overflowing user stack
- Interrupts
  - Devices
- Swapper, pagedaemon

# The Kernel interacts with processes and devices

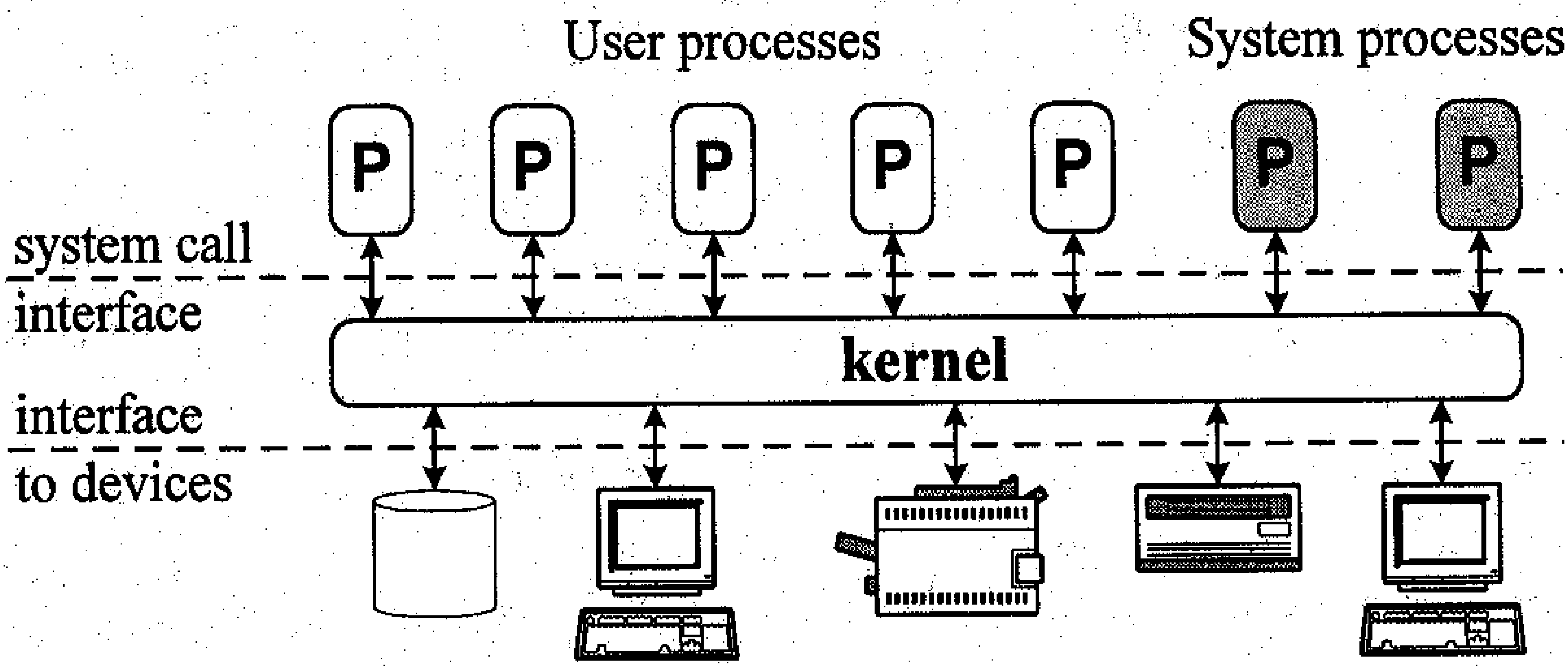



**Figure 2-1.** The kernel interacts with processes and devices.

# Mode,Space & Context

- Some critical resources must be protected
  - Kernel Mode: More privileged, kernel functions
  - User Mode: Less privileged, user functions
- Virtual Memory
  - VM space
  - Address Translation Maps
  - Memory Management Unit

# Kernel data

- Current process & context switch
- One instance of the kernel
- Global data structure
- Per-process objects
- System call, mode switch
- User area: info. about a process
- Kernel stack

# Context

- Re-entrant: several processes may be involved in kernel activities concurrently.
- Execution context
  - Process
  - System (Interrupt )

# Execution mode and Context

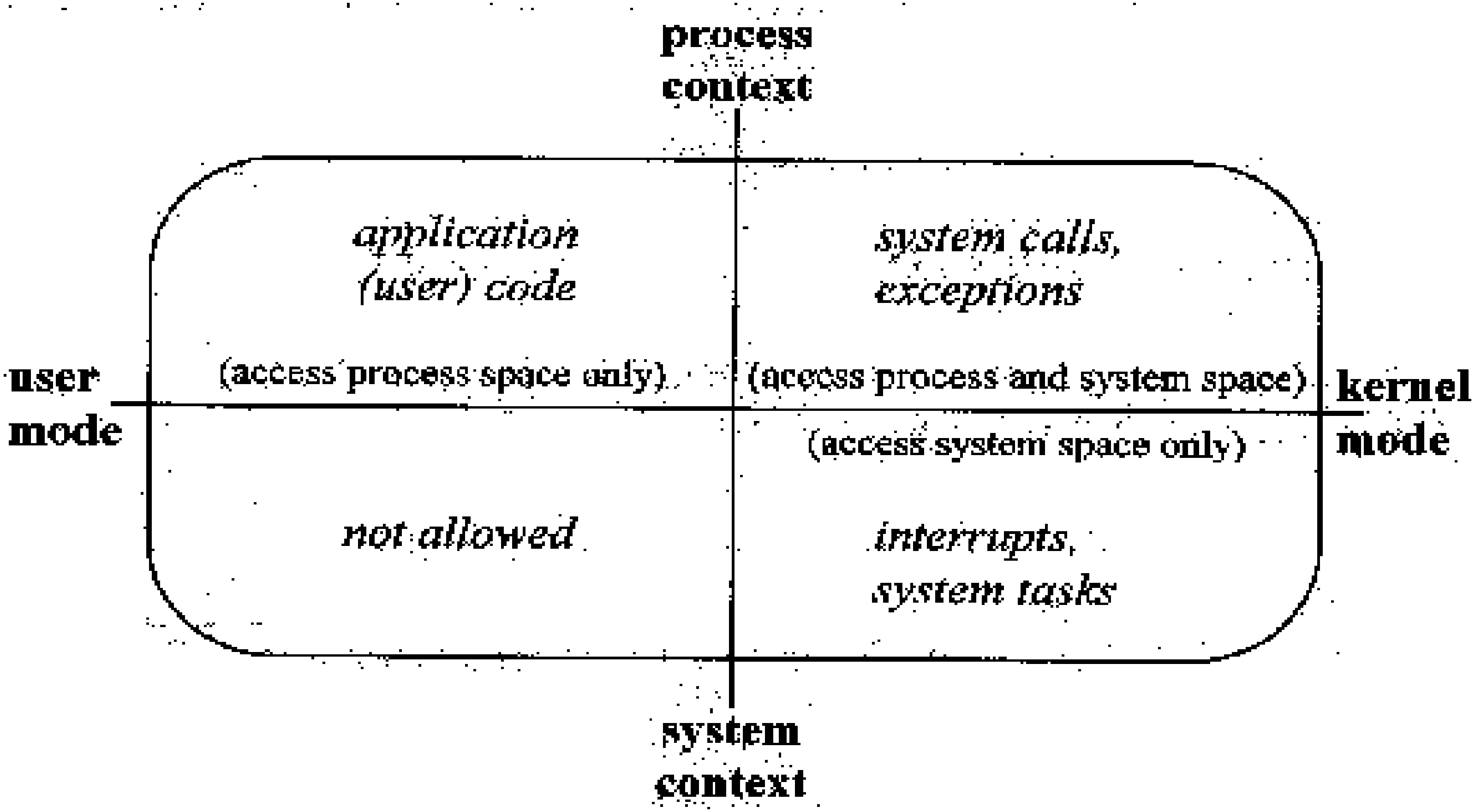



**Figure 2-2.** Execution mode and context.

# Executing in Kernel Mode

- 3 types of events:
  - Device interrupts
  - Exceptions
  - Traps or software interrupts
- Dispatch table
- System context: interrupts
- Process context: traps, exceptions & software interrupts

# The System Call Interface

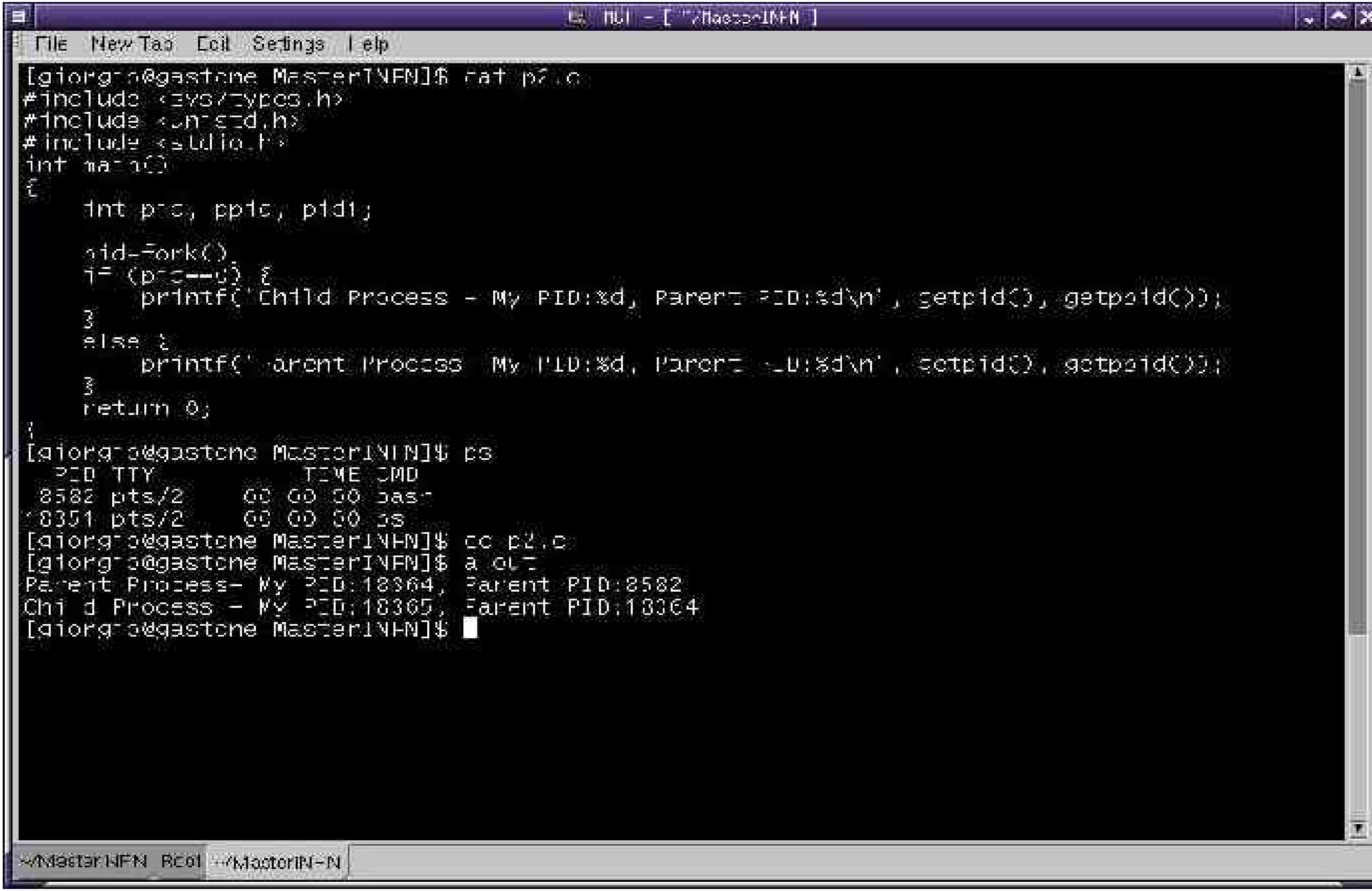
- `syscall()`: the starting point
  - In kernel mode, but in process context.
  - Copy arguments , save hardware context on the kernel stack.
  - Use system call number to index dispatch vector
  - Return results in registers, restore hardware context, to user mode, control back to the library routine.

# New Processes & Programs

- `int fork()`:
  - creates a new process.
  - returns 0 to the child, PID to the parent
- `int exec*(..)`:
  - begins to execute a new program

# Using fork & exec

```
if ((ChildPid = fork())==0){
    /* child code*/
    … …
    if (execve(“new program”),…)<0) {
        • perror(“execve failed.”);
        • exit(-1)
    }
} else if (ChildPid <0) {
        • perror(“fork failed”);
        • exit(-1)
}
/*parent continues here*/
```



```
[giorgio@gastone MasterINFN]$ cat p2.c
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main()
{
    int pid, ppid, pid1;

    pid=fork();
    if (pid==0) {
        printf("Child Process - My PID:%d, Parent PID:%d\n", getpid(), getppid());
    }
    else {
        printf("Parent Process- My PID:%d, Parent PID:%d\n", getpid(), getppid());
    }
    return 0;
}
[giorgio@gastone MasterINFN]$ ps
  PID TTY          TIME CMD
 8582 pts/2    00:00:00 bash
18351 pts/2    00:00:00 ps
[giorgio@gastone MasterINFN]$ cc p2.c
[giorgio@gastone MasterINFN]$ a.out
Parent Process- My PID:18364, Parent PID:8582
Child Process - My PID:18365, Parent PID:18364
[giorgio@gastone MasterINFN]$
```

# Process Creation

- Almost an exact clone of the parent.
  - Reserve swap space for the child
  - Allocate a new PID and proc structure for the child
  - Initialize proc structure
  - Allocate ATM (address translation map)
  - Allocate *u* area and copy
  - Update the *u* area to refer to the new ATM & Swap space
  - Add the child to the set of processes sharing the text region of the program
  - Duplicate the parent’s data and stack regions update ATM to refer to these new pages.
  - Acquire references to shared resources inherited by the child
  - Initialize the hardware context
  - Make the child runnable and put it on a scheduler queue
  - Arrange to return with 0
  - Return the PID to the parent

# Fork Optimization

- It is wasteful to make an actual copy of the address space of the parent
  - Copy-on-write:
    - only the pages that are modified must be copied.(SYSV)
  - vfork() (BSD):
    - The parent loans the address space and blocks until the child returns to it.
      - **dangerous**
    - (csh exploits it)

# Invoking a New Program

- Process address space
  - Text: code
  - Initialized data
  - Uninitialized data(bss)
  - Shared memory(SYSV)
  - Shared libraries
  - Heap: dynamic space
  - User stack: space allocated by the kernel

# Awaiting Process Termination

```
wait(statusp);/* SV, BSD & POSIX*/
wait3(statusp,options,rusagep);/*BSD*/
waitpid(pid,statusp,options);/*POSIX*/
waitid(idtype,id,infop,options);/*SVR4*/
```

# Zombie Processes

- Only holds proc structure.
- `wait()` frees the proc
  - parent or the init process.
- When child dies before the parent & parent doesn't wait for all childs, then the proc is never released.