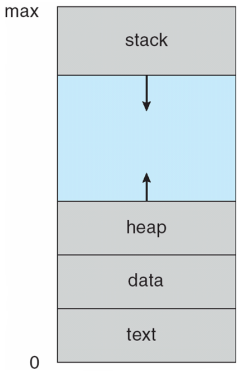
## Management of Application Execution

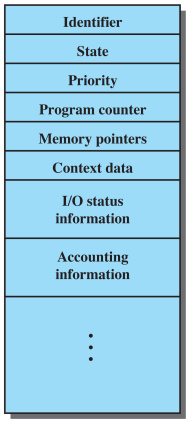
- Resources (processor, I/O devices, etc.) are made available to multiple applications
- The processor in particular is switched among multiple applications so all will appear to be progressing
- Fundamental unit of computation: the process
  - Contains an address space and one or more threads of execution (program counters)
  - Other elements that uniquely characterize a process: identifier, state, priority, context data, I/O status information, accounting information
    = process control block (PCB)






## Process Control Block

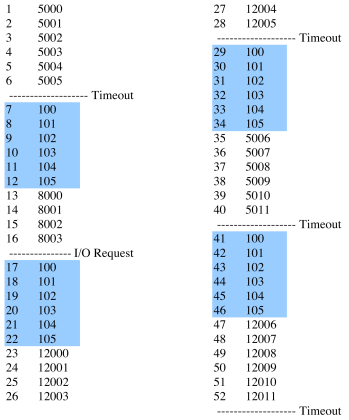
- Contains the process elements
- It is possible to interrupt a running process and later resume execution as if the interruption had not occurred
- Created and managed by the operating system
- Key tool that allows support for multiple processes





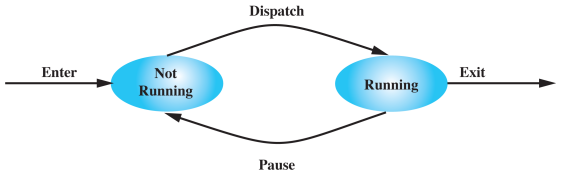

## Process States

- Dispatcher = small prgram that switches the CPU between processes
- Simplest model of execution: the two-state model
  - Program may be in one of two states: running and not running

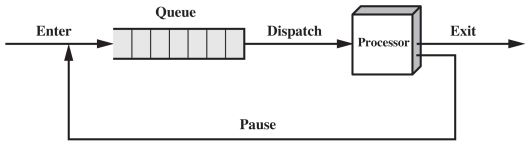





## Two-State Model (cont'd)




(a) State transition diagram




(b) Queuing diagram

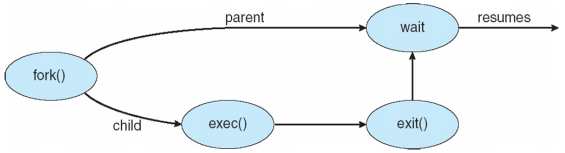
## Process Creation

- Reasons for creating processes:
  - New job
  - New user logon (new process will handle the new session)
  - Created by the OS to provide a service (whenever the user does not need to wait for completion, e.g., a print job)
  - Created by an existing process (for modularity or to exploit parallelism)
- The original process is the parent
- The new process is the child
- We have a tree of processes
- Resource sharing model: share all resources / share a subset of resources / share no resource
- Execution model: parent and child execute concurrently / parent waits until child terminates
- Address space model: child duplicates the parent / child executes new program



## Unix API for Process Creation

- fork() creates new process by duplicating the parent
  - The call to fork() duplicates the current process; both processes continue execution from the instruction following the call to fork()
- exec() replace the process' memory space with a new program
  - The arguments to execve are the name of the command to execute, then two arrays of strings containing the command line arguments and the environment, just as the ones received by the function main.
  - Usually used after fork()






## Process Creation Example

We thus create singly threaded processes.

```
#include <iostream.h>
#include <sys/types.h>
#include <unistd.h>

int main (int argc, char** argv) {
  int i;
  int sum = 0;

  fork();

  for (int i=1; i<10000; i++) {
    sum = sum + i;
  }
    cout << "\nI computed " << sum << "\n";
}
```



## Diverging Processes

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

int main (int argc, char** argv, char** envp) {
  int i;
  int sum = 0;
  int pid;

  pid = fork();

  for (int i=1; i<10000; i++) {
    if (pid == 0)
      printf("+");  // printed by child
    else
      printf("-");  // printed by parent
    sum = sum + i;  // done by both
  }
  if (pid == 0)
    printf("\n[Child] I computed %d\n", sum);
  else
    printf("\n[Parent] I computed %d\n", sum);
}
```

- Processes are identified in Unix by a unique process identifier (or PID for short), which is an integer.
- `fork()` returns two different integers in the child and parent processes.
  - In the child process `fork()` returns zero.
  - In the parent process `fork()` returns the PID of the newly created child.
  - Different code for the parent and the child can then be surrounded in appropriate `if` statements.

## Children Doing Something Completely Different

- The call `execve` replaces completely the current process with another executable. The arguments are the name of the command to execute, then two arrays of strings containing the command line arguments and the environment, just as the ones received by the function `main`
- Suppose now that we want to run an external command (so we use `execve`)...
- ...but we want to continue the execution of the main program too
- We use a combination of `fork` and `execve`:

```
int childp = fork();
if (childp == 0) { // child
  execve(command, argv, envp);
}
else { // parent
  // code that continues our program
}
```



## Know What Your Children Do

- Again, suppose that we want to execute an external command (`execve` again)
- we still want to continue the execution of the main program,
- but only after the external command has been completed

```
int run_it (char* command, char* argv [], char* envp[]) {
  int childp = fork();
  int status;

  if (childp == 0) { // child
    execve(command, argv, envp);
  }
  else { // parent
    waitpid(childp, &status,0);
  }
  return status;
}
```

- Variants of `waitpid` and `execve` also exist



## Process Termination

- Process executes last statement and asks the operating system to delete it (exit)
  - Output data from child to parent (if applicable, via wait)
  - Process' resources are deallocated by operating system
- Parent may terminate execution of children processes (abort)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - Parent is exiting
- Some operating system do not allow child to continue if its parent terminates
  - All children terminated = cascading termination
- Other reasons for termination: segmentation violation, arithmetic error, invalid instruction, memory unavailable, protection error (instructions, I/O), I/O error



## Intermission: Unix Signals

- Processes as well as the OS can send signals to each other
  - Main control mechanism for process termination
  - A signal is just an `int` communicated to a process
  - Most signals are sent automatically by the system cases (e.g., `SIGCHLD`)
  - But can also be sent signals from one's code (the meaning of most signals is defined by the system, but `SIGUSR1` and `SIGUSR2` are user-defined)
  - For a description of available signals, see `man -S7 signal`
- To send a signal, use the function `int kill(pid_t pid, int sig);` (see `man -S2 kill`) where `pid` is the process id of some process, or
  - 0 to send the signal to all the processes in the process group
  - -1 to send the signal to all the processes except the first one (`init`)
- To receive a signal, you should establish a signal handler in your program by using the function `signal` (see `man -S2 signal`).
  - The signal handler fires assynchronously once for each received signal
  - Special signal that cannot be handled: `SIGKILL`

## Process Termination in Unix

- Issue: If we use `fork()` to create new processes, they will not terminate completely (zombie processes)
  - All zombies want is to communicate their exit status
  - A zombie process dies for good when its parent executes `waitpid` on them
  - In fact when a child process terminates, it sends a `SIGCHLD` signal to its parent
  - if we do not want to wait for children we can create a function `reaper` that fires up whenever a `SIGCHLD` signal is received

    We then put before any call to `fork`:

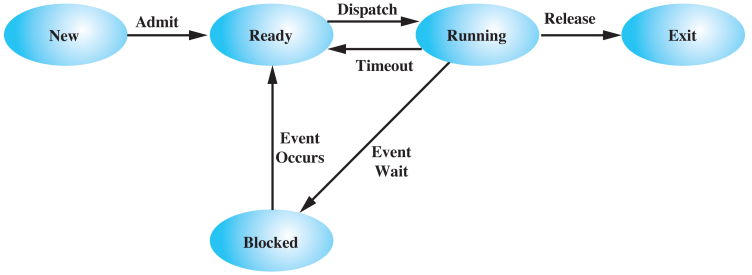
```
signal(SIGCHLD, reaper);
```

```
void reaper (int sig) {
  int status;
  while (waitpid(-1,&status,
                 WNOHANG) >= 0)
    /* NOP */ ;
}
```



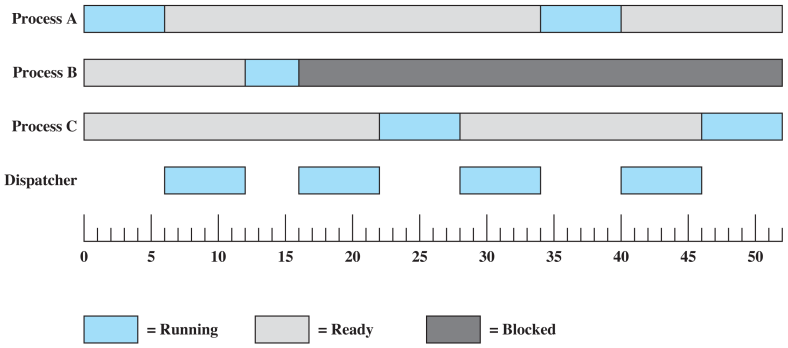
## Five-State Process Model

- The two-state model is inefficient whenever processes perform I/O
- In such a case they should not hold the CPU



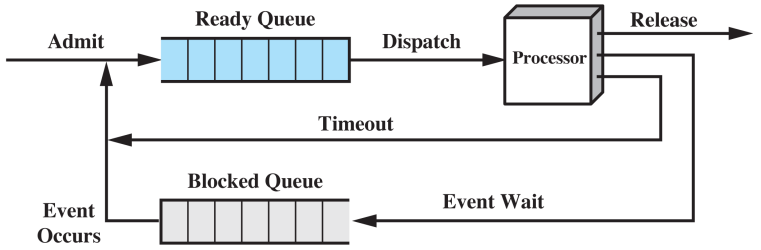



## Five-State Process Model (cont'd)

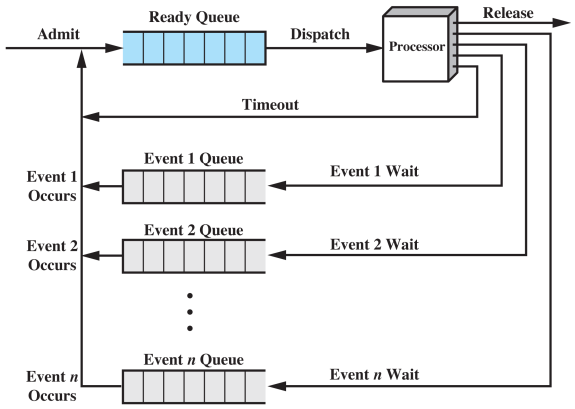





## Five-State Process Model: Single Blocked Queue



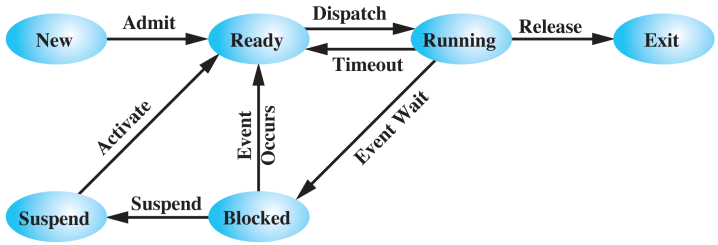
## Five-State Process Model: Multiple Blocked Queues





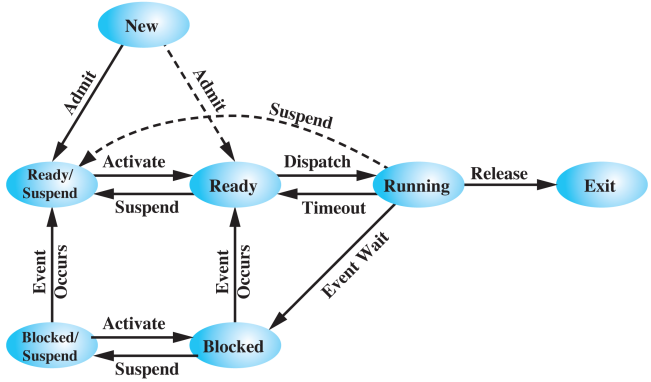

## Suspended Processes

- The OS may place blocked processes into a suspend queue
- Suspended processes may be further swapped i.e., moved to from the main memory to disk






## Suspended Processes: Two Suspend States

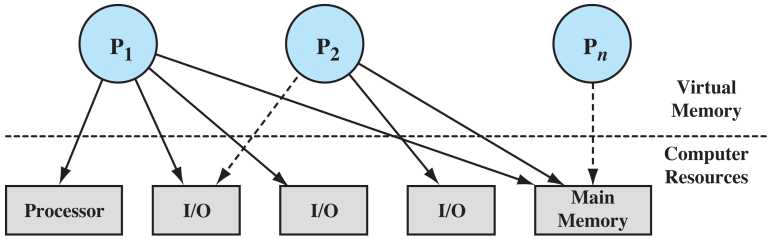





## Suspended Processes (cont'd)

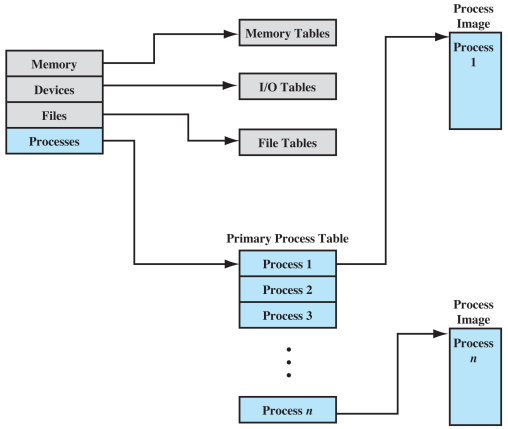
- Characteristics:
  - The process is not immediately available for execution
  - The process was placed in a suspended state by an agent (itself, a parent process, the OS) for the purpose of preventing its execution
  - The process may or may not be waiting on an event
  - The process may not be removed from this state until the agent explicitly orders the removal
- Possible reasons for suspension:
  - Swapping (the OS needs to release main memory to execute a ready process)
  - Other OS reason (e.g., suspend a utility suspected of causing a problem)
  - User request (e.g., debugging reasons)
  - Tming (a periodic process may be suspended while waiting)
  - Parent request (for examination or coordination)
- In Unix a process can be suspended using the `SIGSTOP` or `SIGSTP` signals
  - The difference is that `SIGSTOP` cannot be handled but `SIGSTP` can

## Processes and Resources






## OS Control Tables






## OS Control Tables (cont'd)

- Memory tables are used to keep track of both main (real) and secondary (virtual) memory
  - Processes are maintained on secondary memory using some sort of virtual memory or simple swapping mechanism
  - Must include allocation of main memory to processes, allocation of secondary memory to processes, protection attributes of blocks of main or virtual memory, information needed to manage virtual memory
- I/O tables are used by the OS to manage the I/O devices and channels of the computer system
  - At any given time, an I/O device may be available or assigned to a particular process
  - Must include for any I/O operation in progress the status of the operation and the location in main memory being used as the source or destination of the I/O transfer



## OS Control Tables (cont'd)

- File tables provide information about existence of files, location of secondary memory, current status, other attributes
  - Information may be maintained and used by a file management system (in which case the OS has little or no knowledge of files)
  - In other operating systems, much of the detail of file management is managed by the OS itself
- Process tables must be maintained to manage processes
  - There must be some reference to memory, I/O, and files (directly or indirectly)
  - The tables themselves must be accessible by the OS and therefore are subject to memory management
  - To manage and control a process the OS must know: where the process is located, and its relevant attributes

## Process Control Structures

- Process location
  - A process must include a program or set of programs to be executed
  - A process will consist of at least sufficient memory to hold the programs and data of that process
  - The execution of a program typically involves a stack that is used to keep track of procedure calls and parameter passing between procedures, as well as unstructured space (heap)
- Process image
  - Used by the OS for process control
  - Program + data + stack + attributes = process image
  - Process image location depends on the memory used management scheme



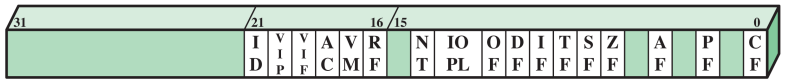
## Process Attributes

Process attributes are stored in the PCB and include the following:

- Process identification
  - Numeric identifiers stored in the PCB
    - process ID: identifies the process uniquely throughout the system (in all the OS tables)
    - parent ID
    - owner (user) ID
- Processor state information
  - User-visible registers
  - Control and status registers: various registers used to control the operation of the CPU, including the program counter, condition codes (result of the most recent arithmetic or logical operation), and status information or program status word or PSW (includes an interrupt enable/disable flag and the execution mode)
- Stack pointers: point to the top of each stack used by the process



## PSW Example: The Pentium EFLAGS Register



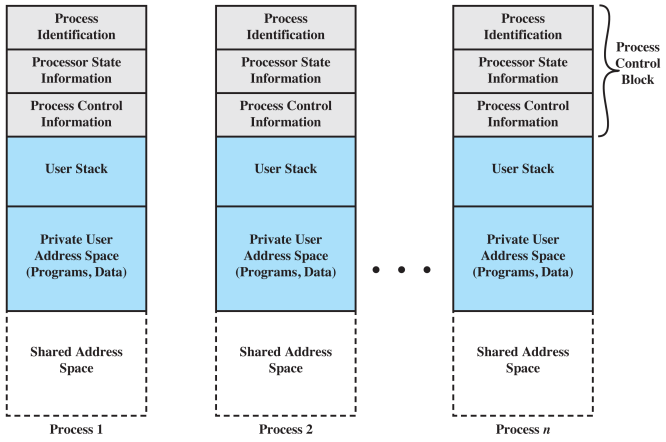

| | | | | | |
|---|---|---|---|---|---|
| ID | = | Identification flag | DF | = | Direction flag |
| VIP | = | Virtual interrupt pending | IF | = | Interrupt enable flag |
| VIF | = | Virtual interrupt flag | TF | = | Trap flag |
| AC | = | Alignment check | SF | = | Sign flag |
| VM | = | Virtual 8086 mode | ZF | = | Zero flag |
| RF | = | Resume flag | AF | = | Auxiliary carry flag |
| NT | = | Nested task flag | PF | = | Parity flag |
| IOPL | = | I/O privilege level | CF | = | Carry flag |
| OF | = | Overflow flag | | | |



## Process Control Block

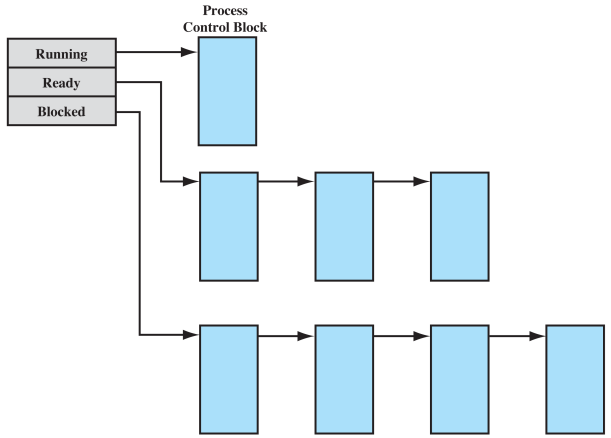
- Additional process control block data
  - Scheduling and state information: state (e.g., running, ready, waiting, etc.), priority, scheduling information (dependent of the used scheduling algorithm), event (the event needed for resuming if any)
  - Data structuring for linking the PCB in various PCB structures such as the process queues
  - Interprocess communication: flags, signals, messages
  - Privileges primarily for memory and instruction sets but also other resources
  - Memory management: pointers to tables that describe the virtual memory assigned to the process
  - Resource ownership and utilization such as opened files (the descriptor table) and utilization history (as possibly needed by the scheduler)
- PCB is the most important data in the whole OS
  - PCBs are read and modified by virtually every OS module
  - Defines the state of the OS
  - Problem is not access, but protection

## Process Images in the Virtual Memory






## Process List Structures






## Process Management

- Process creation: the OS assigns aunique ID, allocates space for the process, initializes the PCB, sets the appropriate PCB linkage, and creates or expands the related data structures
- Process scheduling and dispatching — later
- Process switching: a process switch may occur at any time that the OS has gained control from the running process. Possible events giving such a control include:
  - Interrupts: caused by an event external to the process, used to react to asynchronous external events (in kernel mode)
    - Examples: clock, I/O, memory fault
    - Time slice = the maximum amount of time a process can execute before being interrupted
  - Trap: caused by the execution of the current instruction, used to handle an error or exception condition
  - Supervisor call: caused by an explicit request from the process, used to call an OS function (system calls)



## Process State Change

- Several steps are needed for a full process switch:
  1. Save the context of the processor
  2. Update the process control block of the process currently in the Running state
  3. Move the process control block of this process to the appropriate queue
  4. Select another process for execution
  5. Update the process control block of the process selected
  6. Update memory management data structures
  7. Restore the context of the processor to that which existed at the time the selected process was last switched out
- If the currently running process is to be moved to another state (Ready, Blocked, etc.), then the OS must make substantial changes in its environment

## Security Issues

- An OS associates a set of privileges with each process
- Typically a process that executes on behalf of a user has the privileges that the OS recognizes for that user
- Highest level of privilege is referred to as administrator, supervisor, or root access
- Key security issue in the design of any OS: prevent, or at least detect, attempts by a user or a malware from gaining unauthorized privileges on the system and from gaining root access
- Countermeasures:
  - Intrusion detection = sensors + analyzers + user interface; host or network based
  - Authentication = identification + verification; four general authentication means:

| | |
|---|---|
| something the user knows (password, PIN) | something the user is (static biometrics) |
| something the user possesses (smart card, "token") | something the user does (dynamic biometrics) |

  - Access control = enforces and monitor access and type of access to resources
  - Firewalls for network protection



## Unix System V Release 4

- Uses the model where most of the OS executes within the environment of a user process
- System processes run in kernel mode
  - Executes operating system code to perform administrative and housekeeping functions
- User processes
  - Operate in user mode to execute user programs and utilities
  - Operate in kernel mode to execute instructions that belong to the kernel
  - Enter kernel mode by issuing a system call, when an exception is generated, or when an interrupt occurs

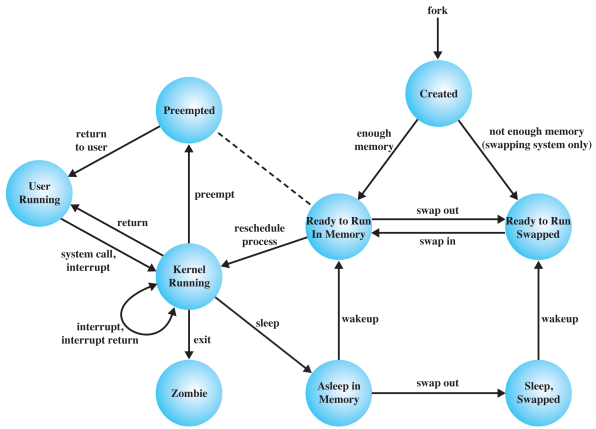


## Unix Process States

| | |
|---|---|
| User Running | Executing in user mode |
| Kernel Running | Executing in kernel mode |
| Ready to Run, in Memory | Ready to run as soon as the kernel schedules it |
| Asleep in Memory | Unable to execute until an event occurs; process is in main memory (a blocked state) |
| Ready to Run, Swapped | Process is ready to run, but the swapper must swap the process into main memory before the kernel can schedule it to execute |
| Sleeping, Swapped | The process is awaiting an event and has been swapped to secondary storage (a blocked state) |
| Preempted | Process is returning from kernel to user mode, but the kernel preempts it and does a process switch to schedule another process |
| Created | Process is newly created and not yet ready to run |
| Zombie | Process no longer exists, but it leaves a record for its parent process to collect |



## Unix Process State Transition Diagram



## UNIX PROCESS IMAGE

| | |
|---|---|
| | **User-Level Context** |
| Process text | Executable machine instructions of the program |
| Process data | Data accessible by the program of this process |
| User stack | Contains the arguments, local variables, and pointers for functions executing in user mode |
| Shared memory | Memory shared with other processes, used for interprocess communication |
| | **Register Context** |
| Program counter | Address of next instruction to be executed; may be in kernel or user memory space of this process |
| Processor status register | Contains the hardware status at the time of preemption; contents and format are hardware dependent |
| Stack pointer | Points to the top of the kernel or user stack, depending on the mode of operation at the time or preemption |
| General-purpose registers | Hardware dependent |
| | **System-Level Context** |
| Process table entry | Defines state of a process; this information is always accessible to the operating system |
| U (user) area | Process control information that needs to be accessed only in the context of the process |
| Per process region table | Defines the mapping from virtual to physical addresses; also contains a permission field that indicates the type of access allowed the process: read-only, read-write, or read-execute |
| Kernel stack | Contains the stack frame of kernel procedures as the process executes in kernel mode |



## UNIX PROCESS TABLE ENTRY

| | |
|---|---|
| Process status | Current state of process. |
| Pointers | To U area and process memory area (text, data, stack). |
| Process size | Enables the operating system to know how much space to allocate the process. |
| User identifiers | The real user ID identifies the user who is responsible for the running process. The effective user ID may be used by a process to gain temporary privileges associated with a particular program; while that program is being executed as part of the process, the process operates with the effective user ID. |
| Process identifiers | ID of this process; ID of parent process. These are set up when the process enters the Created state during the fork system call. |
| Event descriptor | Valid when a process is in a sleeping state; when the event occurs, the process is transferred to a ready-to-run state. |
| Priority | Used for process scheduling. |
| Signal | Enumerates signals sent to a process but not yet handled. |
| Timers | Include process execution time, kernel resource utilization, and user-set timer used to send alarm signal to a process. |
| P_link | Pointer to the next link in the ready queue (valid if process is ready to execute). |
| Memory status | Indicates whether process image is in main memory or swapped out. If it is in memory, this field also indicates whether it may be swapped out or is temporarily locked into main memory. |



## UNIX U AREA

| | |
|---|---|
| Process table pointer | Indicates entry that corresponds to the U area. |
| User identifiers | Real and effective user IDs. Used to determine user privileges. |
| Timers | Record time that the process (and its descendants) spent executing in user mode and in kernel mode. |
| Signal-handler array | For each type of signal defined in the system, indicates how the process will react to receipt of that signal (exit, ignore, execute specified user function). |
| Control terminal | Indicates login terminal for this process, if one exists. |
| Error field | Records errors encountered during a system call. |
| Return value | Contains the result of system calls. |
| I/O parameters | Describe the amount of data to transfer, the address of the source (or target) data array in user space, and file offsets for I/O. |
| File parameters | Current directory and current root describe the file system environment of the process. |
| User file descriptor table | Records the files the process has opened. |
| Limit fields | Restrict the size of the process and the size of a file it can write. |
| Permission modes fields | Mask mode settings on files the process creates. |



## UNIX PROCESS CREATION

- First process: PID 0, with process image hard coded in the kernel; launches the init process
- Init (PID 1) responsible for launching all the other processes
- Process creation:
  1. Allocate a slot in the process table for the new process
  2. Assign a unique process PID to the child process
  3. Make a copy of the process image of the parent, with the exception of any shared memory
  4. Increments counters for any files owned by the parent, to reflect that an additional process now also owns those files
  5. Assigns the child process to the Ready to Run state
  6. Returns the PID number of the child to the parent process, and a 0 value to the child process
- After creation the kernel can dispatch as follows: stay in parent, switch to child, or switch to another process

## UNIX PROCESS SECURITY

- Each user is assigned a unique user ID and can belong to any number of groups (identified by unique group IDs); UID 0 and GID 0 are reserved for root
- Every process inherits its parent's UID and GIDs
  - Privileged processes (UID 0) can drop privileges by changing their UID/GIDs but cannot get them back once dropped
- Once the user logs in he gets a first process (login shell) with his UID and GIDs. Every process launched by the user thus inherits the respective UID and GIDs
  - Access granted based on the current UID and GIDs
  - The file system has a triple set of permissions (user, group, others) that control access
  - Access to other resources (printers, peripherals, configuration changes, etc.) is controlled via the file system (that is, they are accessed via virtual files or directories)
  - Privileged access can be given to users selectively by suid executables (have the UID of the file rather than the user who launches them) and sgid executables (have the GID of the file, not the user)