**The Process**

Previous systems were batch systems which used to execute **jobs**.

**Job** – refers to unit of work or tasks submitted to OS for execution. They could comprise of multiple processes.

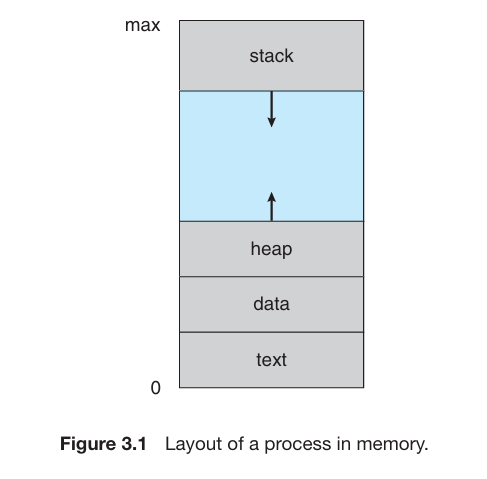
**Process** – refers to a program in execution.

The status of current process in execution is determined by **program counter** and by the contents of processes’ register.

**Program Counter** – Register in the CPU which holds the address of next instruction to be executed.

The memory layout of a process is typically divided into multiple sections:

* **Text Section** – the code to be executed
* **Data Section** – global variables
* **Heap** –memory dynamically allocated during program runtime
* **Stack** – temporary data storage when invoking functions (parameters, local variables, return addresses)



The stack and heap grow as dynamically during execution. An **activation record** is maintained for each function. Every time a function is called activation record is pushed onto the stack. After the function has completed execution the activation record is popped from the stack.

**Activation Record** – collection of parameters, local variables and return address of an individual function.

The heap grows when the memory is allocated and shrinks when returned to the system. Although heap and stack grow towards each other it is the responsibility of OS to prevent an overlap.

|  |  |
| --- | --- |
| **Program** | **Process** |
| A *passive* entity like a file containing a list of instructions sitting on a disk (also called **executable file**). | *Active* entities, with a program counter specifying the next instruction to be executed. |

A program becomes a process when an executable file is loaded into the memory. Two common ways to load an executable is; by double clicking the executable file or using a command.

Although two processes may be associated with the same program, they are nevertheless considered two separate execution sequences. For instance, several users may be running different copies of the mail program, or the same user may invoke many copies of the web browser program. Each of these is a separate process; and although the text sections are equivalent, the data, heap, and stack sections vary. It is also common to have a process that spawns many processes as it runs.

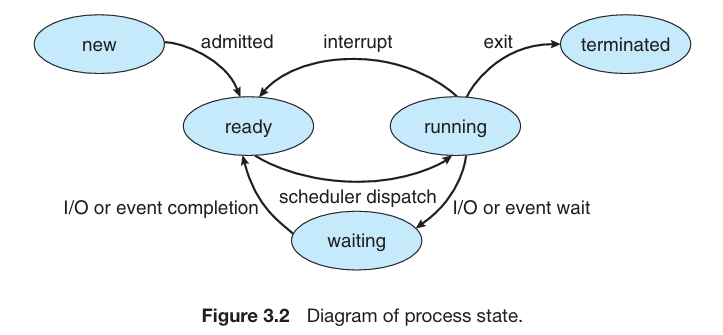
Note that a process can itself be an execution environment for other code. For example, the Java Virtual Machine (JVM) executes an executable java program. It runs a process which interprets the code and takes actions on behalf of the code.

**Process State**

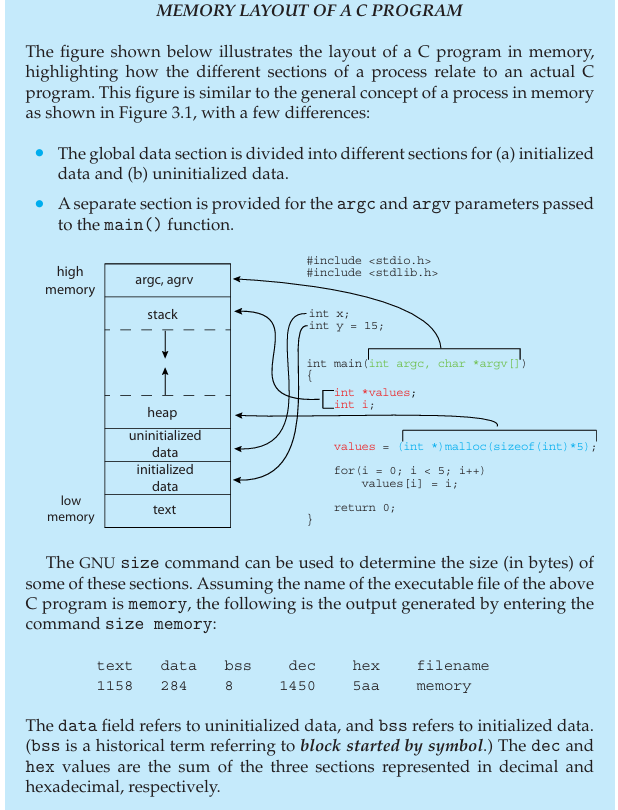
**State** – refers to the current activity of the process.

When a process executes, it changes states. It can be in one of the following states:

* **New** – a new process is being created
* **Running** – instructions are being executed
* **Waiting** – process is waiting for an event to occur like a I/O completion or reception of signal
* **Ready** – the process is ready to be assigned to the processor
* **Terminated** – the process has finished execution



Only one process is running at a time in a processor core at an instant. Others are ready or waiting.



**Process Control Block**

It is a data structure used by the OS to store information about a process necessary for managing and controlling its execution. The said information includes:

**Process State** – could be ready, waiting, running or any other state

**Program Counter** – stores address of next instruction to be executed for the process

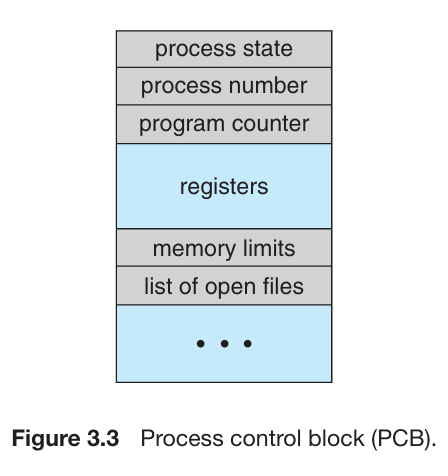
**CPU registers** – common ones include accumulator, general purpose, stack pointer and index registers. Number of registers depends on the computer architecture. Additional code info is also stored here. Program counter and CPU registers are saved before an interrupt is handled so that the execution can begin from where it was halted.

**CPU-scheduling information** – includes process priority, pointers to scheduling queues and any other scheduling parameters.

**Memory management ­**­– includes info about the value of base and limit registers, page tables, segment tables depending on the memory used by system.

**Accounting information** – includes information about CPU and real time used time limits, job or process numbers, account numbers and so on.

**I/O status information** – includes information about the I/O devices allocated to the process, list of open files and so on.



**Threads**

So far a process runs a single **thread** of execution is a fact which has been established.

**Thread** – smallest unit of execution within a process is a thread. It shares the same memory space and resources of the process they belong to.

A single thread of control allows process to perform only one task at a time. But the process concept has been extended by many modern OS which allow a process to run multiple threads of execution allowing them to perform multiple tasks at a time. In multicore processors multiple threads can run in parallel. Due to this the PCB is also enhanced to hold information about multiple threads.

**Process scheduling**

As we know the main goal of multiprogramming is to keep at least one process running at all times so that the CPU is utilized fully. The main objective of time sharing is to switch the CPU so frequently so that the user can interact with each process while it is running.

To achieve this, the **process scheduler** selects a process from a set of available processes to run at a CPU core/program execution at a CPU core. Each CPU core can run one process at a time.

**Process scheduler** – responsible for allocating system resources to a process

If there are more cores than processes than they’ll have to wait until a core is free and can be rescheduled.

**Degree of multiprogramming** – refers to the number of processes running currently in the memory

General behavior of the process must be considered when balancing the objectives of multiprogramming and time switching

Most processes are defined as either *I/O bound* or *CPU-bound*.

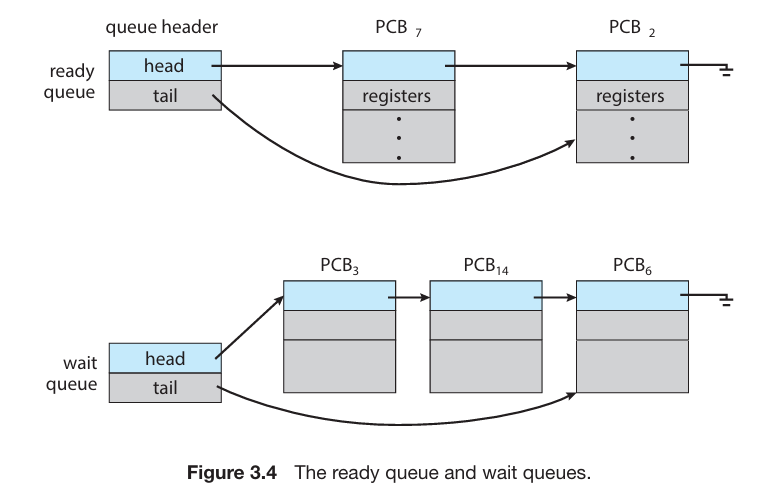
|  |  |
| --- | --- |
| **I/O Bound** | **CPU-bound** |
| Spends most of its time doing I/O than computations | Frequently generates I/O requests, using most of its time doing computations |

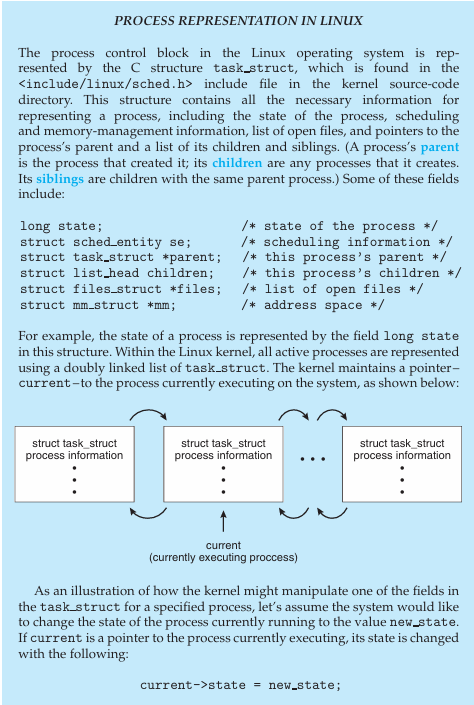
**Scheduling Queues**

As processes enter the system, they are held in a **ready queue** where they are ready and waiting to be executed in one of CPU’s cores.

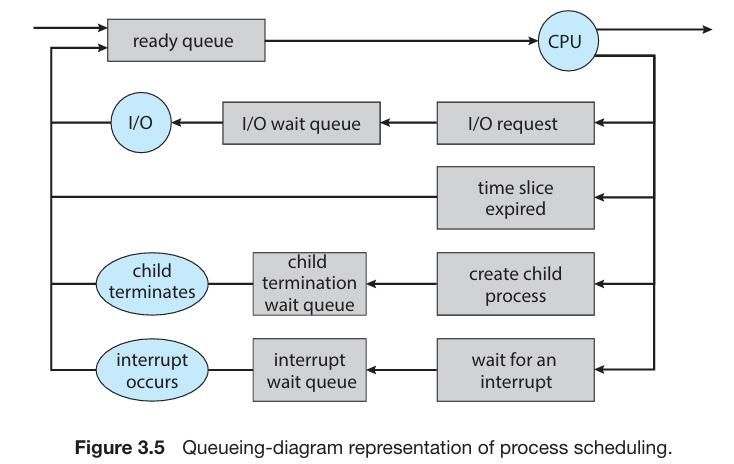
**Ready queue** – It is a data structure (linked-list) which holds the processes ready for execution. The head points to the first PCB in the list and each PCB has a pointer which points to the next PCB in the list.

When a process is allocated a core, it executes and is either terminated, interrupted or waits for a particular event. This event could be reading from a file or I/O event. Processes which wait for such events are placed in the **wait queue**.





A common representation of process scheduling is a *queuing diagram*. The circles represent the resources which serve the processes and arrows represent the flow.



A new process is placed in the ready queue where it is eventually selected for execution. During its execution the following could occur:

* The process could issue an I/O request and then be placed in an I/O wait queue
* The process could create a new child process and then be placed in a wait queue while it awaits the child’s termination.
* The process could be removed forcibly from the core, as a result of an interrupt or having its time slice expire, and be put back in the ready queue.

In the first two cases, the process switches eventually from waiting to ready state and then is placed back in the ready queue. This cycle continues until the process is terminated. When terminated the process is removed from all queues and its PCB and resources are de-allocated.

**CPU scheduling**

**CPU scheduler** – responsible for deciding which processes are or threads are allocated CPU time and in what order. It selects a process from the ready queue and allocates a CPU core to it. It must select new process for the CPU frequently.

Although CPU-bound processes require CPU core for longer period of time, it is unlikely the scheduler allows it. It forcibly removes the CPU core from the process and allocates a new process to it. Hence, the scheduler executes every 100 milliseconds or even more frequently.

Some OS use a form of scheduling called *swapping*. Main idea behind swapping is that a process for the time being is removed from the memory and can later be reintroduced where it will start execution from where it left off. This reduces the degree of multiprogramming. It is usually used where memory has been overcommitted and needs to be freed up.

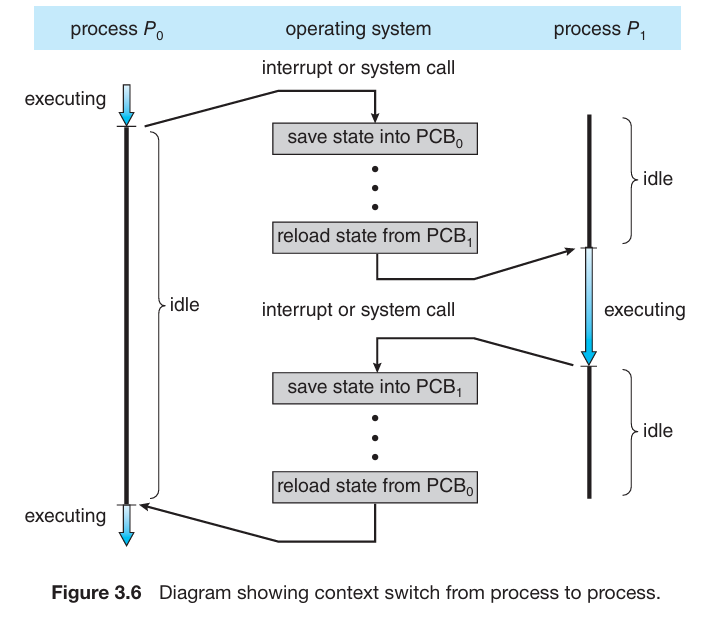
**Context switch**

Takes place when an interrupt is fired/asserted causing the CPU to halt its current operation and run a kernel routine to deal with the interrupt. The context of the process running during which the interrupt was asserted is saved by the system. When the interrupt is handled then the process is resumed, essentially the process is suspended for some time and then resumed from where it was suspended.

The context is represented in the PCB of the process. It includes the process state, CPU register values and memory management info. Generally a *state save* of a CPU core is performed when the interrupt is asserted and after it is handled a *state restore* is performed restoring the CPU core to its previous state.

**Context switch** - Switching the CPU core to another process requires performing a state save of the current process and a state restore of a different process. This task is known as a context switch.

The kernel saves the state of the current process in its PCB and loads the saved context of the new process that is scheduled to run. This switch time is complete overhead as the system does no useful task during it. Switching speeds vary by system. They depend on the memory size, number of registers to be copied and special instructions.



Context-switch times are highly dependent on hardware support. If a system processor has multiple set of registers, then the context switch here would mean simply changing the pointer to the current register set. If there are more active processes than registers then the system copies register data to and from memory. More complex the OS, more work needs to be done for a context switch.

Advanced memory-management techniques may require that extra data be switched with each context. For instance, the address space of the current process must be preserved as the space of the next task is prepared for use. How the address space is preserved, and what amount of work is needed to preserve it, depend on the memory management method of the operating system.

**Operations on Processes**

Processes can run concurrently on modern systems. They can be deleted or created dynamically. The system provides some mechanism for such operations.

**Process creation**

During execution, a process may create several more processes. The creating process is called *parent process*. The processes which emerge from parent are called *child processes*. Each of these newly created processes may create more processes forming a process tree.

**Process identifier (PID)** – it is a unique ID given to an individual process by the OS. It is an integer which can be used as an index to refer various attributes of a process.

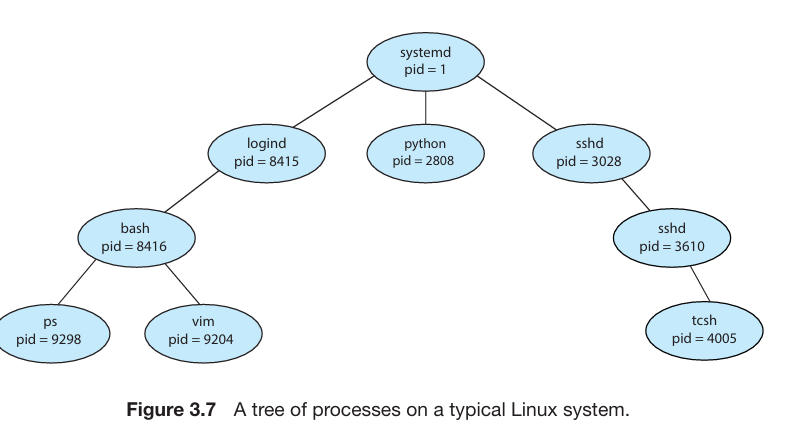
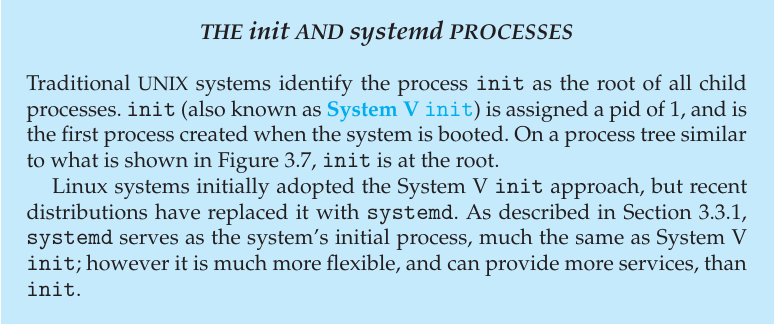


Fig 3.7 represents a process tree in Linux. The root of the tree is *systemd.* It will always have a PID of 1. It has 3 children – *logind, python, and sshd.* Systemd is the first user process created when the system is booted. Once the system boots, systemd creates other child processes which provide services like a print server or Ssh server. Logind is responsible for managing clients directly logged onto the system. The sshd process is responsible for managing clients that connect to the system using Ssh (secure shell).



When a child process is created, it requires some resources. It may obtain them from the OS or will be restricted to its parent’s resources. The parent may have to divide it’s resource amongst the children or share some of them with the children. Restricting a child process to a subset of parent’s process avoids creating overload on the system as it prevents parent from creating too many child processes.

Parent may pass along data initialization values to the child along with physical and logical resources. For example, consider a process whose function is to display the contents of a file— say, hw1.c—on the screen of a terminal. When the process is created, it will get, as an input from its parent process, the name of the file hw1.c.Using that filename, it will open the file and write the contents out. It may also get the name of the output device.

Some OS pass resources to the child process. On such systems, child processes get to open the file and the terminal device and transfer datum between them.

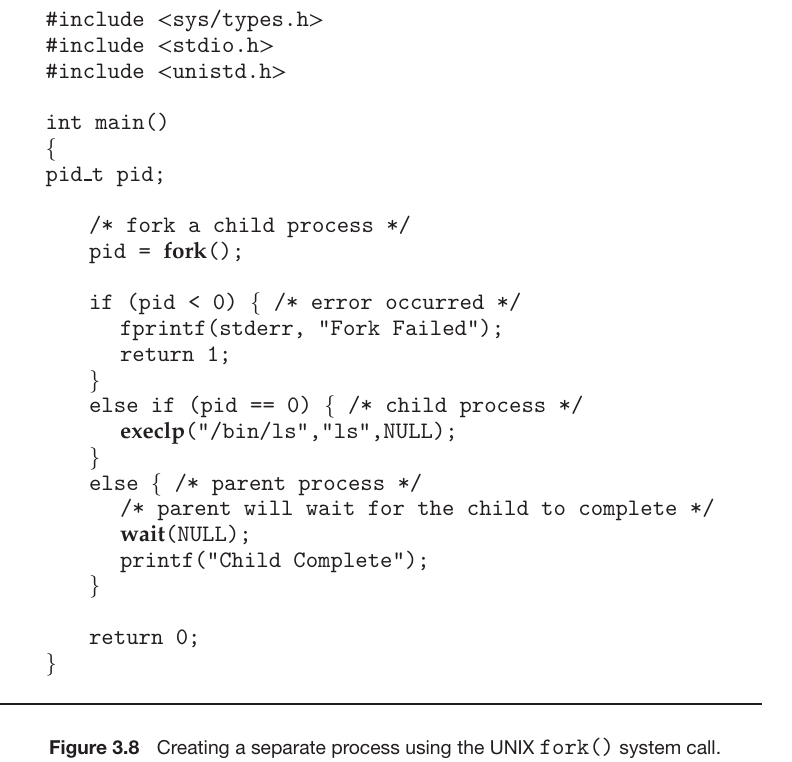
When a process creates another process, 2 possibilities for execution exist:

1. Concurrent execution with the parent executing side by side with its child
2. Parent waits until some or all children are eliminated.

There are also 2 address space possibilities:

1. Child is a duplicate of the parent.
2. Child has a new program loaded into it.

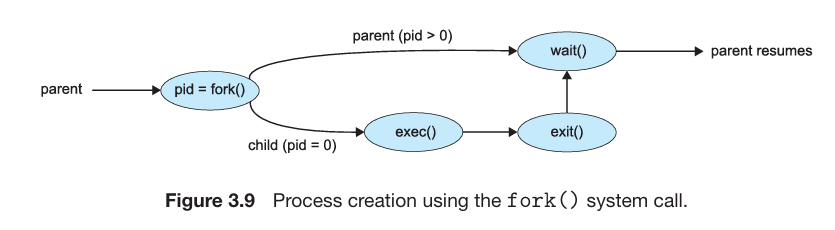
A new process is created by the fork () system call. The new process consists of a copy of the address space of the original process. This mechanism allows the parent process to communicate easily with its child process. Both processes (the parent and the child) continue execution at the instruction after the fork (), with one difference: the return code for the fork () is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent.



After a fork () system call, one of the two processes typically uses the exec () system call to replace the process’s memory space with a new program. The exec () system call destroys the memory image of the process which contains the exec () system call. It loads a binary file in its memory and starts execution.

In this manner, the two processes are able to communicate and then go their separate ways. The parent can then create more children; or, if it has nothing else to do while the child runs, it can issue a wait () system call to move itself off the ready queue until the termination of the child. Because the call to exec () overlays the process’s address space with a new program, exec () does not return control unless an error occurs.

When the child process completes (by either implicitly or explicitly invoking exit ()), the parent process resumes from the call to wait (), where it completes using the exit () system call. There is no way to stop child from not invoking exec () and instead continuing to execute as a copy of the parent process. In this scenario, the parent and child are concurrent processes running the same code instructions. Because the child is a copy of the parent, each process has its own copy of any data.



**Process termination**

After executing its final instruction/statement, the process terminates. It terminates by asking the OS to terminate it via the exit () system call. AT that time the process may return a status value (an integer) to its waiting process through the wait () system call. All resources allocated are de-allocated by OS after a process terminates.

Only the parent process can invoke a termination system call. Otherwise, a user or a misbehaving application could cause the process to terminate. The parent needs ids of the newly spawned processes to keep track of them, thus the id number of processes is passed to the parent.

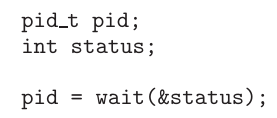
A parent may terminate execution of its child due to:

1. Child exceeding the allocated resources.
2. Child has finished its execution.
3. The parent process is ending and the OS doesn’t allow child to run if the parent terminates.

**Cascading termination** – If a process terminates, all its children are also terminated. This phenomena is called cascading termination.

We can terminate a process by using the exit () system call. The exit () system call can also have a parameter like exit (1). It can be called directly or indirectly – C – library invokes exit () by default.

Parent processes uses wait () system call to wait for its children to finish terminate. This wait () is passed a parameter which allows the parent to obtain the exit status of the child. Wait () returns the id of the process telling the parent which of its process has been terminated.



When a process terminates, its resources are de-allocated by the operating system. However, its entry in the process table must remain there until the parent calls wait (), because the process table contains the process’s exit status.

**Zombie** – It is a process which has been terminated but its parent hasn’t called wait ().

**Orphans** – Processes which are still running but their parent has been terminated are called orphans.

To solve the orphan problem, UNIX based systems allocate systemd as the new parent of the running child processes. The systemd periodically calls wait (), allowing the status of processes to be collected and its id and table entry to be released.

**Android process hierarchy**

Due to limited resources android systems employ *importance hierarchy* in which the more important processes causes another process to terminate to make resources available to itself. This is done by Android OS.

The importance hierarchy is as follows:

1. **Foreground process** – Current process running on the screen like a user application.
2. **Visible process** – A process not directly visible but is performing activity which the foreground process is referring to.
3. **Service process** – Performing an activity apparent to the user like music streaming.
4. **Background process** – A process performing an activity not apparent to the user.
5. **Empty process** – A process which holds no active component related to any application.

If system resources must be reclaimed, Android will first terminate empty processes, followed by background processes, and so forth. Processes are assigned an importance ranking, and Android attempts to assign a process as high a ranking as possible.

Furthermore, Android development practices suggest following the guide lines of the process life cycle. When these guidelines are followed, the state of a process will be saved prior to termination and resumed at its saved state if the user navigates back to the application.

**Inter-process Communication**

There are two types of processes running concurrently in the operating system:

|  |  |
| --- | --- |
| **Independent** | **Cooperative** |
| An independent process does not share data with other processes running in the system. | A cooperative process shares data with another process and is affected by another process running in the system. |

Why allow inter-process communication? Well there are several reasons for this:

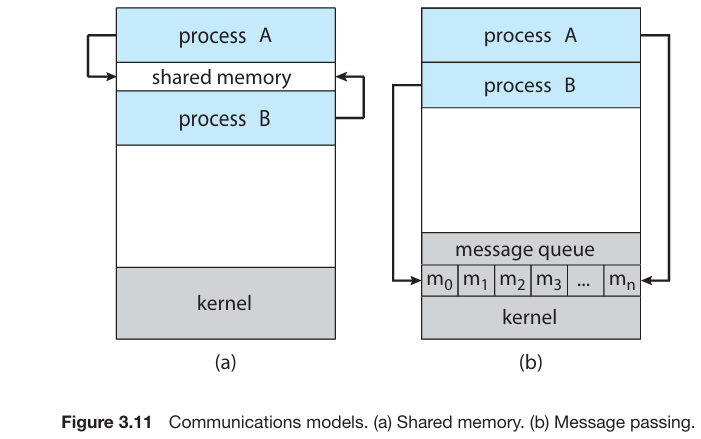
* **Information sharing** – Some applications may be interested in the same piece of information. We must provide an environment so that these applications can access information concurrently.
* **Computation speed** – If we want a task to execute quickly, we may break it down into sub parts which will run concurrently with others. This can only be done on multicore systems.
* **Modularity** – If we want our OS to be broken down into separate modules i.e. dividing system functions into separate processes.

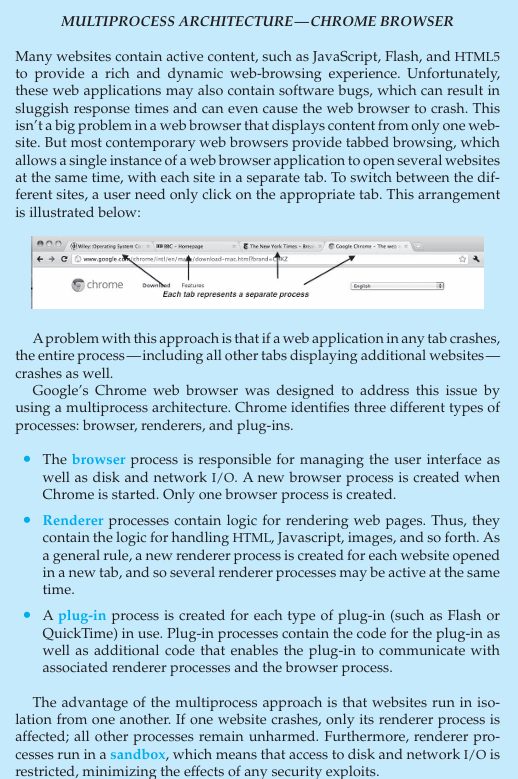
Cooperating process need some sort of functionality or mechanism that’ll allow them to communicate. This is where **inter process communication (ICP)** comes in.

**Inter-process communication** – It is a mechanism which allows processes to exchange data (send or receive) from each other.

There are two fundamental models of inter-process communication:

|  |  |
| --- | --- |
| **Shared memory** | **Message passing** |
| In shared memory, a region of memory that is shared between processes is established. The processes read and write data to the shared region by exchanging information.  Conflicts could occur if two processes try to write or read data from the same address at the same time (concurrently).  Faster than message passing because kernel assistance is not required. Once the region of shared memory is established through system call, all access is treated as routine access.  Not utilized by many distributed systems. | In message passing, processes communicate through messages exchanged between the processes.  It is useful when dealing with small amounts of data because no conflicts need to be avoided.  Slower than shared memory as kernel intervention is required and is implemented by system calls.  Easier to implement in distributed systems. |





**IPC in shared memory systems**

As discussed before, IPC require processes to establish a region of shared memory. Shared memory region resides in the same address space of the process creating the shared memory segment. Other processes who wish to communicate through this segment must attach it to their address space.

But the OS prevents processes from accessing each other’s memory. So in order to communicate, the processes would need to remove this restriction. Shared memory requires two or more processes agree to remove this restriction. The processes decide where the shared region will be located and what form of data will be shared. All of this is out of operating system’s control. Processes also need to ensure that they’re not writing the same location simultaneously.

**Producer-consumer**

It illustrates the concept of cooperating processes. A **producer** process produces informationconsumed by the **consumer**.

**Producer** – It is a program or a component in a system which generates information, typically with the purpose of passing it to another process. In producer-consumer paradigm, it is responsible for creating and supplying data to the shared region.

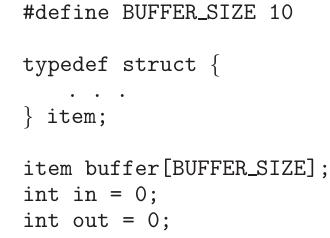
**Consumer** – It is a program or a component in a system which retrieves and utilizes data produced from processes. In producer-consumer paradigm, it is responsible for retrieving and processing data from the shared region.

Producer and consumer must be synchronized in way such that a consumer does not try to consume an item not produced yet.

One solution to the producer-consumer problem is shared memory. To allow producer and consumer processes to run concurrently, a buffer of item is required. This buffer will be filled by the producer and emptied by the consumer. It will reside in the shared memory region of the producer consumer processes.

|  |  |
| --- | --- |
| **Unbounded** | **Bounded** |
| In an unbounded buffer there is no limit on the buffer size  Producer can keep producing. The consumer may have to wait if the buffer is empty. | In a bounded buffer a fixed buffer size is assumed  Producer must wait if the buffer is full and the consumer must wait if the buffer is empty. |

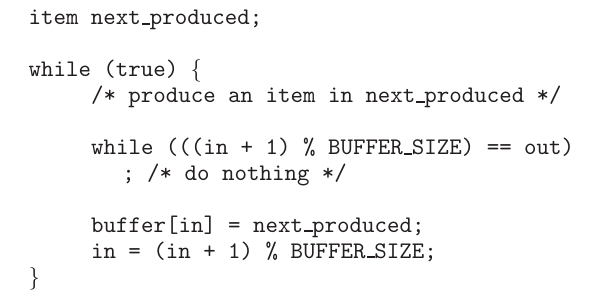
The following code illustrates a bounded buffer:



The shared buffer on the other hand is implemented as a circular array with two pointers; *in* and *out*. The in variable points to the next free position in the buffer while the out variable points to the next full position.

There are two conditions to check:

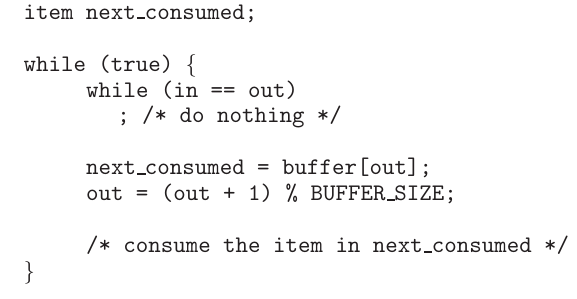
1. If in is equal to out (in == out), this means the buffer is empty.
2. If (in + 1) % size == out, this means the buffer is full. Here the *size* is the buffer size.



The above code represents a producer producing a process. Its breakdown is as follows:

* A variable next\_produced is declared with type item (structure defined with attributes). It holds the new item to be produced.
* There are two while loops, the second one being nested.
* The purpose of the second while is to check if the buffer is full, if it is full, wait for an item to be consumed.
* The new item produced held in the next\_produced variable is written inside the buffer.
* The in pointer is updated to point to the next free position in the buffer.

This method allows size – 1 process in the buffer at the same time. Another issue is that the above process does not address the situation where the producer and consumer try to access the buffer concurrently.



The above code illustrates the consumer process. Its breakdown is as follows:

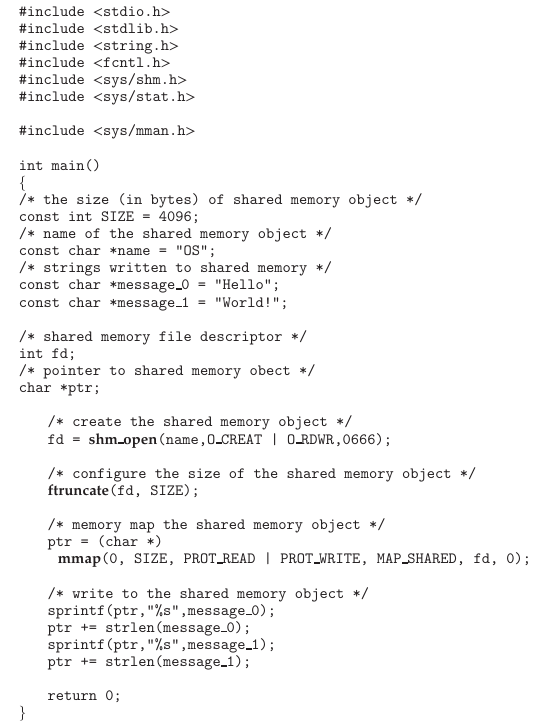
* A variable next\_consumed of item type is declared. It will hold the item to be consumed.
* There are two while loops. The inner while loop (nested) checks if the buffer is empty. If it is then it waits for an item to be produced which then it can consume.
* Item to be consumed is read from buffer and is stored in the next\_consumed variable.
* After item consumption, the out pointer is updated to the next item to be consumed/next full position in the buffer.

**POSIX shared memory**

In POSIX systems the shared memory is organized by memory mapped files. Here the region of shared memory is associated with a file.

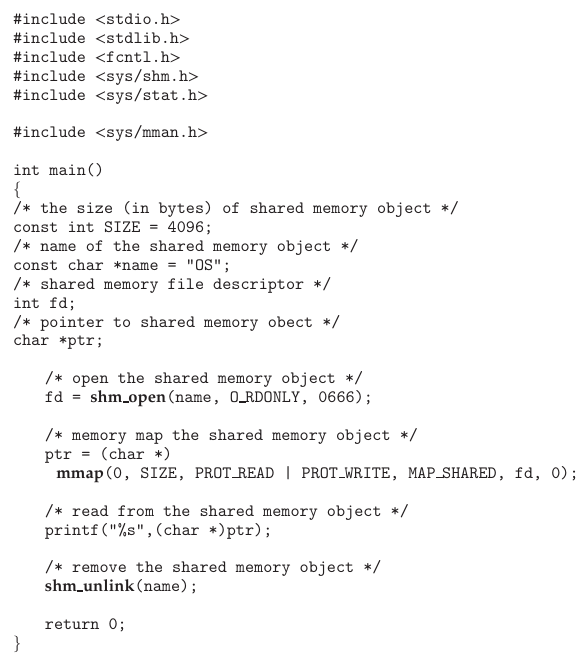
The process of creating the shared file object along with memory allocation and mapping the file is described below.

* A process must create a shared memory object using shm\_open () system call. It takes three parameters; name of the object, the mode in which the object is to be opened or created if it doesn’t exist and file access permissions of the shared memory object.
* The variable fd – short for File Descriptor, contains the value returned by the shm\_open () system call. A successful call will return an integer file descriptor for the object.
* After object has been established, the ftruncate () is used to configure the size of object in bytes. It takes two parameters; the file descriptor and the size. So if it called like; ftruncate (fd, 4096), it means the object has been allocated 4,906 bytes.
* Finally, the mmap () establishes a memory-mapped file containing the shared memory object. It also returns a pointer to the memory-mapped file that is used to access the shared memory object.
* To remove the shared memory object after consumption, shm\_unlink () system call is used. It takes the object name as a parameter and removes the shared memory segment after the consumer has accessed it.



The above program illustrates a producer process using POSIX API. It creates a shared-memory object named OS and writes the infamous string "Hello World!" to shared memory. The program memory-maps a shared-memory object of the specified size and allow writing to the object. The flag MAP SHARED specifies that changes to the shared-memory object will be visible to all processes sharing the object.

Data is written to the shared memory object by sprintf. After each write, we must increment the pointer by the number of bytes written.



The consumer process illustrated above reads and outputs the con tents of the shared memory. The consumer also invokes the shm\_unlink () function, which removes the shared-memory segment after the consumer has accessed it.

**Pipes**

**Pipe** – A pipe acts as a conduit allowing two processes to communicate.

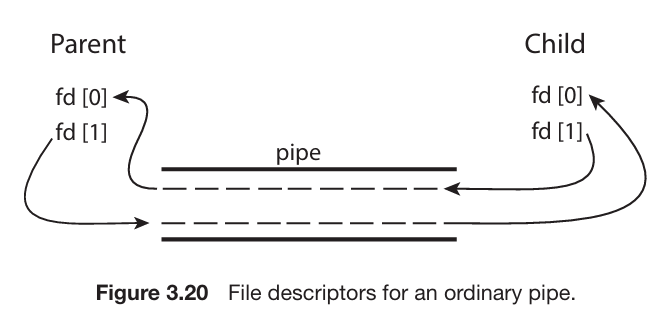
**Ordinary pipe**

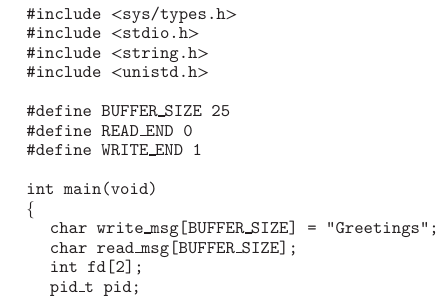
They follow a standard producer-consumer procedure; producer writes at the write-end of the file whereas the consumer reads the data from the other end of the file (the read end). Hence ordinary pipes are unidirectional.

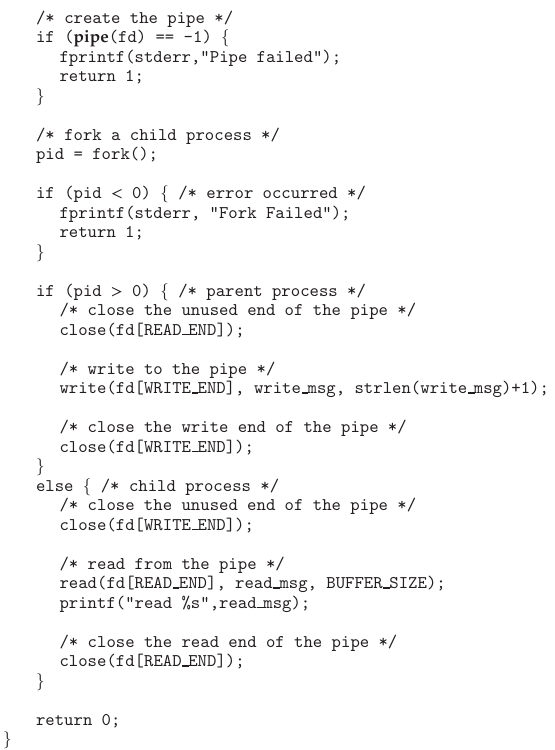


Above method illustrates how pipes are created in UNIX systems. They are a special type of files, hence they can be accessed through read and write system calls. The pipe function takes an fd array. It is typically of size 2. The 0th index specifies the read end of the pipe and the 1st index represents the write end of the pipe.

An ordinary pipe 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 creates via fork (). Since a pipe is a special type of file, the child inherits the pipe from its parent process.







In the UNIX program shown above, the parent process creates a pipe and then sends a fork () call creating the child process. What occurs after the fork () call depends on how the data are to flow through the pipe. In this instance, the parent writes to the pipe, and the child reads from it. It is important to notice that both the parent process and the child process initially close their unused ends of the pipe.

The Windows API for creating pipes is the CreatePipe () function, which is passed four parameters. The parameters provide separate handles for (1) reading and (2) writing to the pipe, as well as (3) an instance of the STARTUPINFO structure, which is used to specify that the child process is to inherit the handles of the pipe. Furthermore, (4) the size of the pipe (in bytes) may be specified.

**Named pipes**

Named pipes provide a much more powerful communication tool. Communication can be bidirectional, and no parent–child relationship is required. Once a named pipe is established, several processes can use it for communication.

In a typical scenario, a named pipe has several writers. Additionally, named pipes continue to exist after communicating processes have finished.

Named pipes are referred to as FIFOs in UNIX systems. Once created, they appear as typical files in the file system. A FIFO is created with the mkfifo () system call and manipulated with the ordinary open (), read (), write (), and close () system calls. It will continue to exist until it is explicitly deleted from the file system.

|  |  |
| --- | --- |
| **UNIX** | **Windows** |
| In UNIX based systems only half duplex transmission is permitted. If data needs to travel in both directions, two FIFOs are used. Additionally, the communicating processes must reside on the same machine. If inter-machine communication is required, sockets must be used.  Only byte-oriented data may be transmitted across a UNIX FIFO | Named pipes on Windows systems provide a richer communication mechanism than their UNIX counterparts. Full-duplex communication is allowed, and the communicating processes may reside on either the same or different machines.  Windows systems allow either byte- or message-oriented data. |

Named pipes are created with the CreateNamedPipe () function and a client can connect to a named pipe using ConnectNamedPipe (). Communication over the named pipe can be accomplished using the ReadFile () and WriteFile () functions.