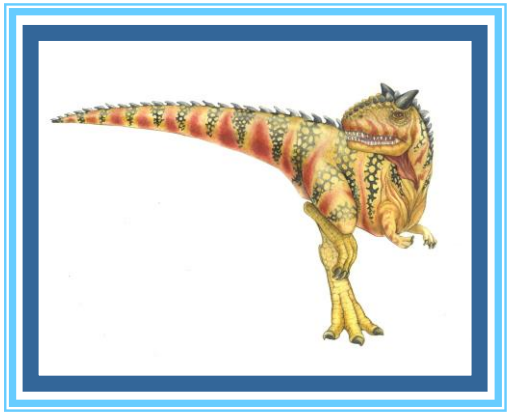
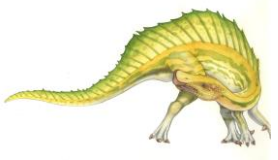
Chapter 3: Processes



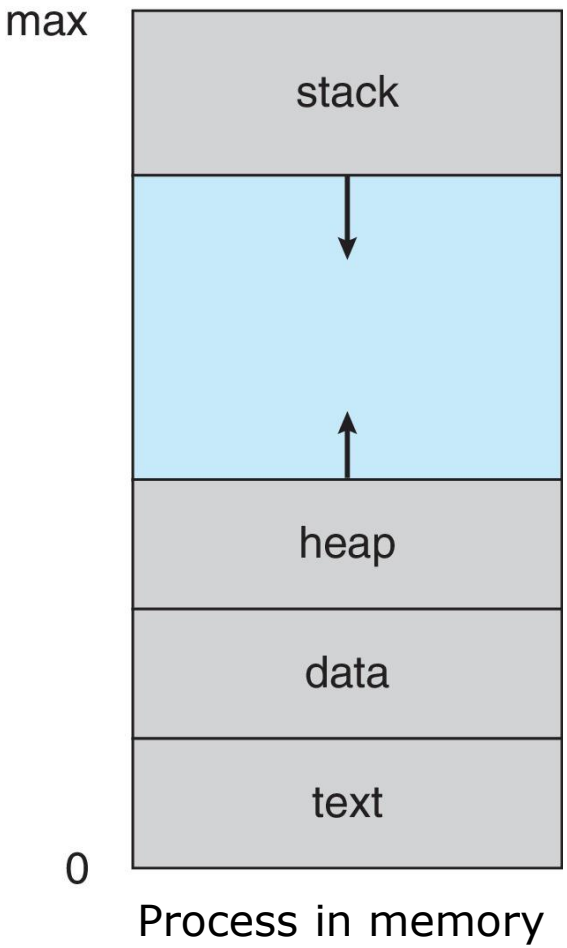
Process Concept

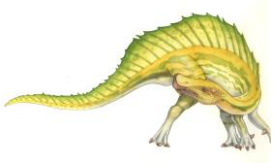
- A **process** is a program in execution
- program vs process
  - A program is a ***passive*** entity stored on disk (**executable file**)
  - A process is an ***active*** entity
  - A program becomes a process when executable file is loaded into memory



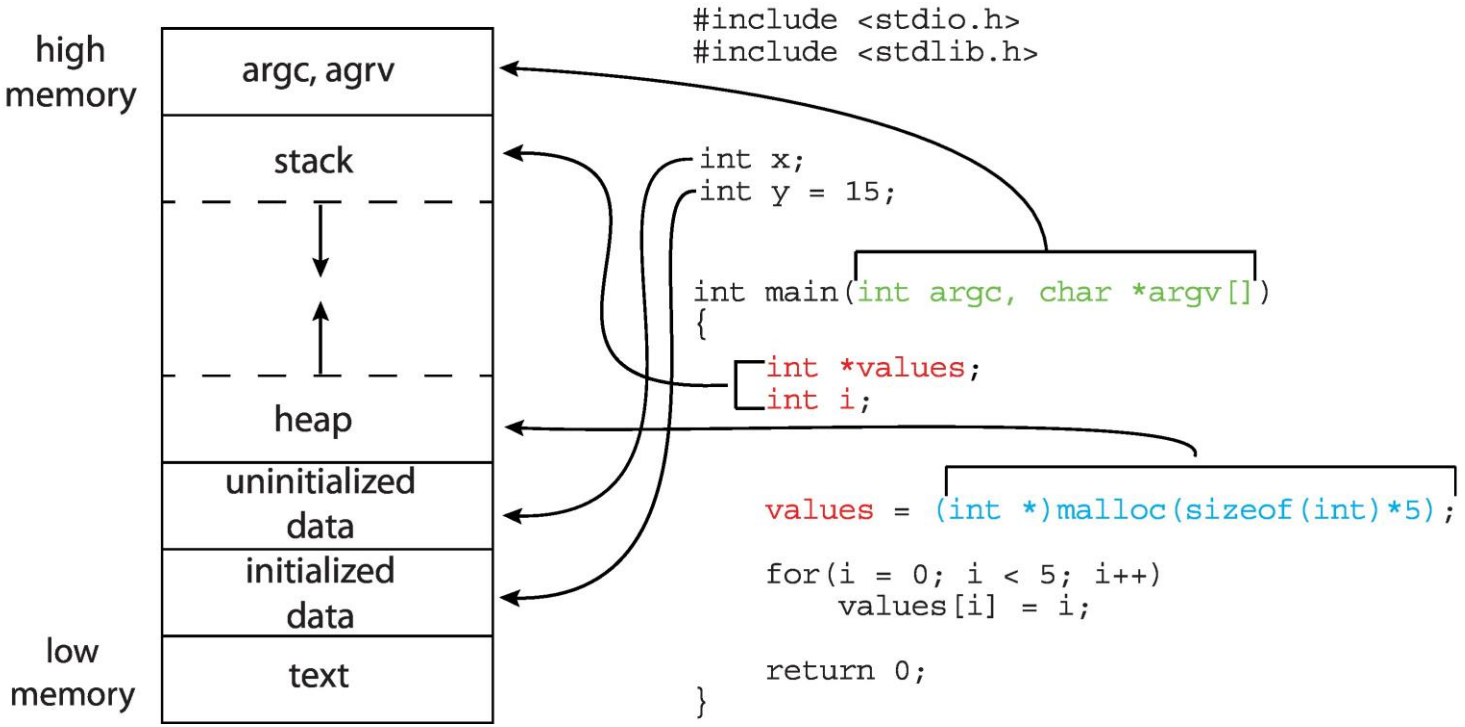
Parts of a Process

- The address space of a process consists of
  - **Text section** containing the program code
  - **Data section** containing global variables
  - **Stack** containing temporary data when invoking functions
    - parameters, return addresses, local variables
  - **Heap** containing memory dynamically allocated during run time
- Current activity of a process represented by
  - Value of **program counter** (i.e., address of next instruction to execute)
  - Content of CPU registers



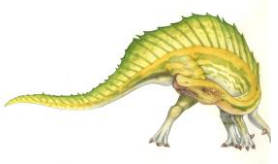


Memory Layout of a C Program



Python Tutor link

Size a.out

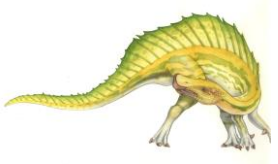


Relationship between Processes and Programs

- Multiple processes may be associated with the same program
  - The processes are separate execution sequences
  - The processes have the same text section
  - The data, heap, and stack sections vary
  - Value of program counter and contents of CPU registers also vary

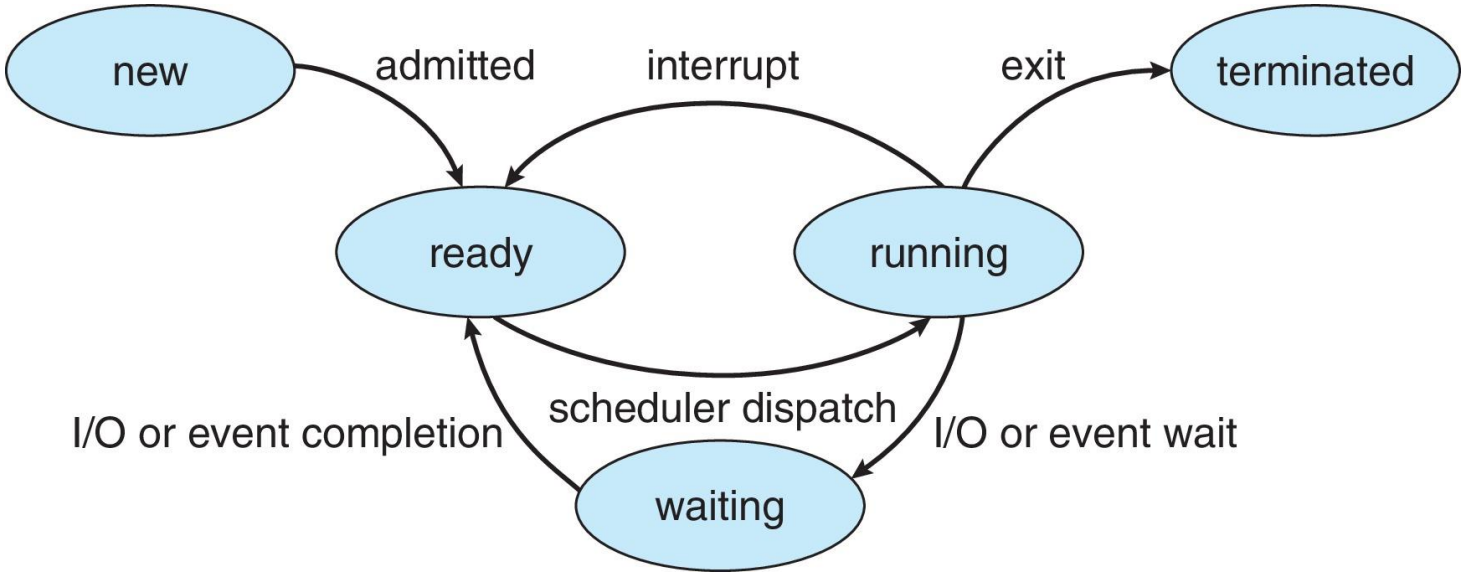
Process State

- As a process executes, it changes **state**
  - **New**: The process is being created
  - **Running**: Instructions are being executed
  - **Waiting**: The process is waiting for some event to occur (e.g., I/O completion)
  - **Ready**: The process is waiting to be assigned to a processor
  - **Terminated**: The process has finished execution
- Only one process can be **running** on a processor at any instant
- Many processes may be **ready** and **waiting**



State Transitions

A **state transition** for a process is a change in its state caused by the occurrence of some event

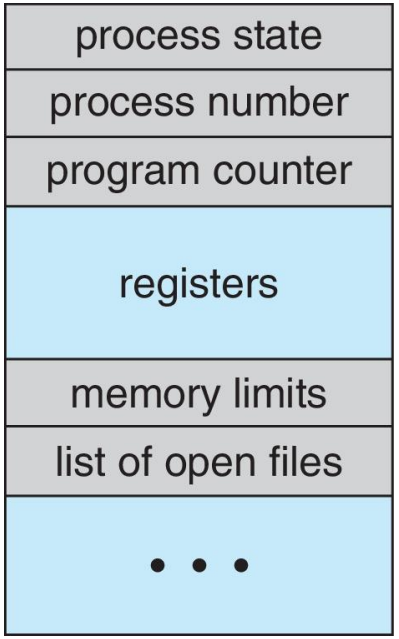


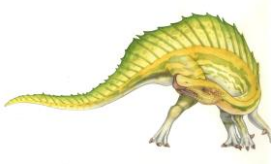
Show Animation 3.1

Process Control Block (PCB)

Process control block (PCB) stores information associated with each process

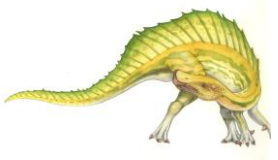
- **Process state** – running, waiting, etc
- **Process number** – unique ID assigned to process
- **Program counter** – address of next instruction to execute
- **CPU registers** – contents of all CPU registers
- **CPU scheduling information-** priorities, scheduling queue pointers
- **Memory-management information** – memory allocated to the process
- **Accounting information** – CPU time used, clock time elapsed since start, time limits
- **I/O status information** – I/O devices allocated to process, list of open files





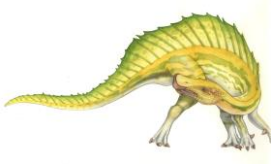
Process Scheduling

- Operating systems use multiprogramming to maximize CPU utilization
- Operating systems use multitasking to enable users to interact with their programs
  - OS switches a CPU core among processes frequently
- **Process scheduler** selects among **ready** processes for next execution on CPU core
- OS maintains **scheduling queues** of processes
  - **Ready queue** – set of processes residing in main memory, ready and waiting to execute
  - **Wait queues** – set of processes waiting for an event (i.e. I/O completion)

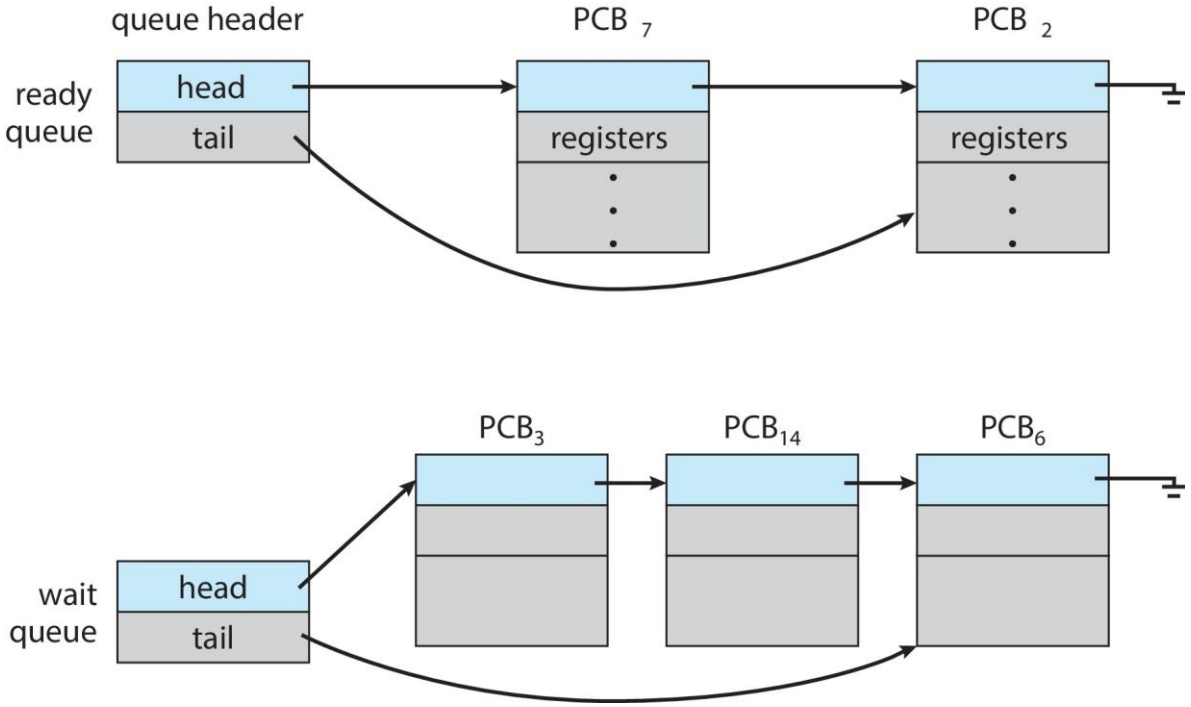


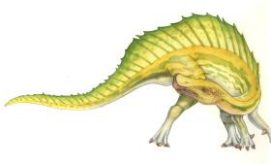
Process Representation in Linux

- linux/sched.h
- Struct task_struct*
- Doubly linked list

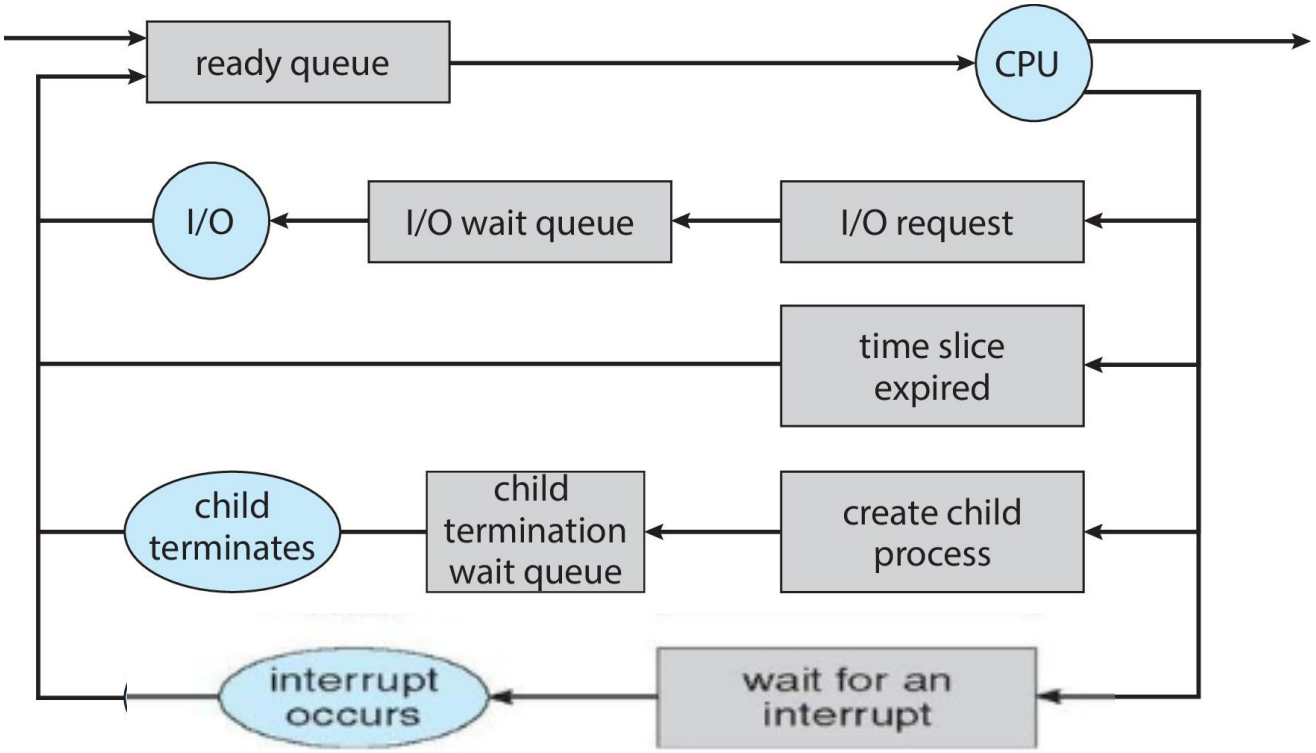


Ready and Wait Queues





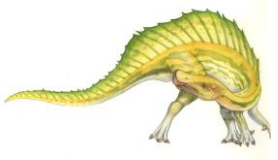
Representation of Process Scheduling



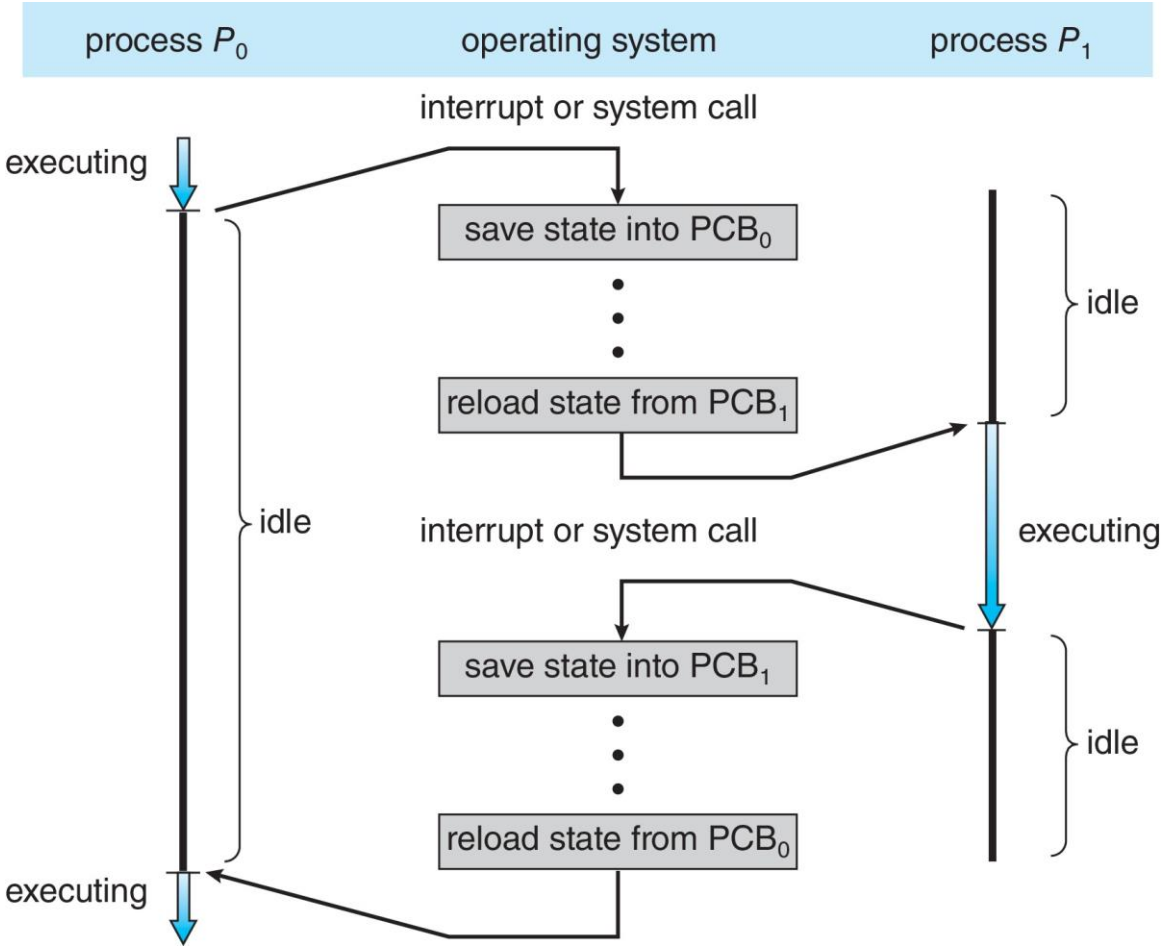
Processes migrate among the queues

Context Switch

- When CPU switches to another process, the system must **save the context** of the old process and load the **saved context** for the new process via a **context switch**
- **Context** of a process saved in its PCB
  - Value of CPU registers, program counter, memory management-information
- Context-switch time (typically a few microseconds) is **overhead**; the system does no useful work while switching



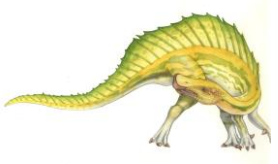
CPU Switch From Process to Process



Show Animation 3.2

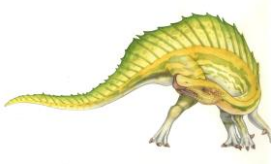
CPU Efficiency in Multitasking Systems

- In a multitasking system, CPU switches rapidly between processes
- A **time slice**, denoted by δ, is the largest amount of CPU time any process can consume when scheduled to execute on the CPU
- If time slice elapses before the process completes its execution, kernel preempts the process
- Let σ denote the context-switch time, then CPU efficiency = $\delta / (\delta + \sigma)$



Operations on Processes

- OS must provide mechanisms for:
  - process creation
  - process termination

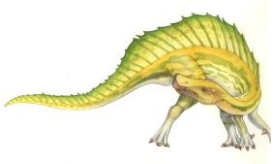


Process Tree in Linux

- Show ps –el
- Process tree all the way upto system (eg: ps)
- https://en.wikipedia.org/wiki/Systemd
- Figure 3.7: Linux Process Tree

Process Creation

- **Parent** process creates **child** processes, which, in turn create other processes, forming a **tree** of processes
  - Each process identified by a **process identifier** (**pid**)
- Resource sharing options
  - Parent and children share all resources
  - Child shares subset of parent's resources
  - Parent and child share no resources
- Address-space possibilities for the child process:
  - Child is a duplicate of parent (child has same program and data as parent)
  - Child has a new program loaded into it
- Execution options
  - Parent and child execute concurrently
  - Parent waits until child terminates



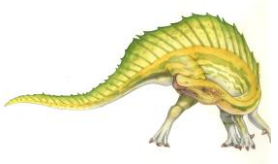
fork() system call

#include <sys/types.h>

#include <unistd.h>

pid_t fork(void); // create a child process identical to parent

- Child process has a copy of parent's address space
- On success:
  - Both parent and child continue execution at the instruction after fork()
- On failure:Returns pid of the child process to the parent process; returns 0 to child process
  - Child is not created; returns –1 to parent

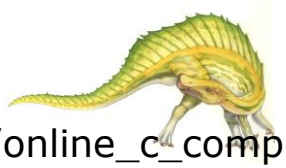


fork() example

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main()
{
      int x=5;
      pid_t pid = fork();
      if (pid < 0) return -1;
      printf("I am running\n");
      if (pid) {
            printf("I am the parent\n");
            x *= 2;
      }  else {
            printf("I am the child\n");
            x += 2;  }
      printf("x is %d\n", x);
      return 0;
}
```

Output

```
P: I am running
P: I am the parent
P: x is 10
C: I am running
C: I am the child
C: x is 7
```



https://www.onlinegdb.com/online_c_compi

Fork() Example 2

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main()
{
int x=5;
pid_t pid = fork();
if (pid < 0) return -1;
printf("I am running\n");
if (pid) {
                x = 10;
                pid_t pid = fork();
                printf("I am the parent\n");
                 x *= 2;
}  else {
                printf("I am the child\n");
                x += 2;  }
printf("x is %d\n", x);
return 0;}
```

```
P0: I am running
P1: I am running
P1: I am the child
P1: x is 7
P2: I am the parent
P2: x is 20
P0: I am the parent
P0: x is 20
```

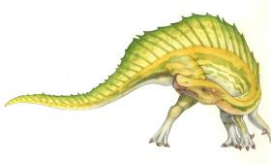
Process Termination (1)

- Process executes last statement and then asks the OS to delete it using the **`exit()`** function.
  - Process' resources (e.g., memory, open files) are deallocated by OS

#include <stdlib.h>

void exit(int status); //cause normal process termination

- The function does not return
- Parent calls **`wait()`** to obtain status value from child



Process Termination (2)

- Parent process may wait for termination of a child process by using the **`wait()`** system call
- **`wait()`** returns status information and the pid of the terminated process

```
pid = wait(&status); //status is an integer pointer
```

- When a process terminates, its PCB must remain until parent calls **`wait()`** as process' exit status is stored in its PCB
- If a process has terminated, but parent has not yet called **`wait()`**, then process becomes a **zombie**
- If parent terminated without invoking **`wait()`**, then the process becomes an **orphan**
  - UNIX assigns the **`init`** process as the new parent to orphan processes; **`init`** process periodically invokes **`wait()`**
  - In Linux, **`systemd`** process becomes the parent to orphan processes

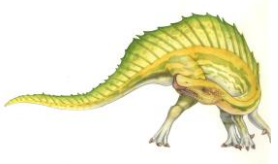
Waiting for Process Termination (1)

#include <sys/types.h>

#include <sys/wait.h>

pid_t wait(int *status) – wait for any child to terminate

- If **status** is not NULL, the exit status of child is stored in the location pointed to by **status**
- On success, returns the pid of the child that terminated; on error, -1 is returned



Waiting for Process Termination (2)

#include <sys/types.h>

#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *status, int options) – wait for the specified child to terminate

- Options
  - 0: wait until child has exited
  - WNOHANG: return immediately if child has not exited
- On success, returns the pid of the terminated child or zero if WNOHANG is used and the specified child has not terminated
- On error, -1 is returned

Note: **wait(&status)** is equivalent to **waitpid(-1, &status, 0)**

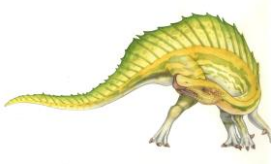
wait() example

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

int main()
{
  pid_t pid = fork();
  if (pid < 0) return -1;
  if (pid==0) {
        printf("I am the child\n");
        exit(0);
  }
  waitpid(pid, NULL, 0);
  printf("Child has terminated, so will I\n");
  return 0;
}
```

Output

C: I am the child
P: Child has terminated, so will I



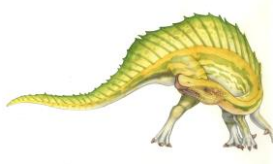
Executing Another Program from a Process (1)

#include <unistd.h>

int execlp(const char *file, const char *arg, ...)

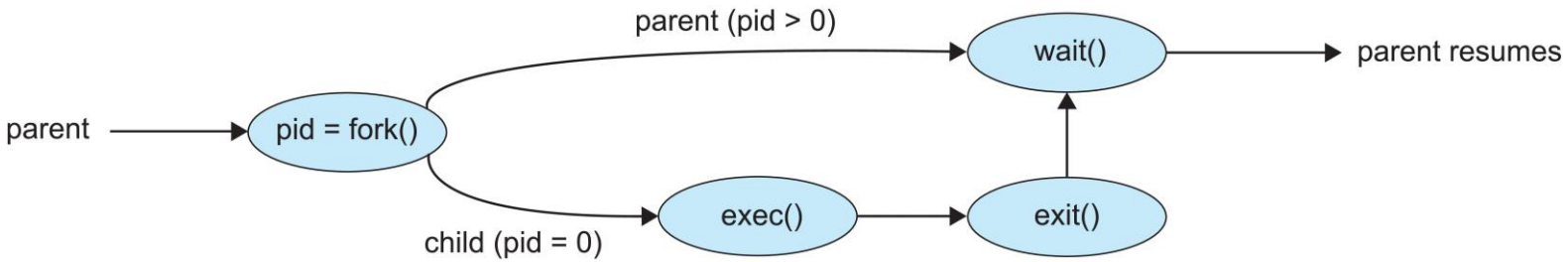
int execvp(const char *file, char *const argv[ ])

- The functions are used to replace the current process image with a new process image.
- The first argument is the name of a file that is to be executed
- **const char *arg, …** in execlp() describes a list of one or more pointers to null-terminated strings that represent the argument list to the executed program
- **char *const argv[ ]** in execvp() is an array of pointers to null-terminated strings that represent the argument list to the executed program
- The first argument in argument list/array should point to the filename associated with the file to be executed
- The argument list/array must be terminated by a NULL pointer
- The functions do not return on success and return –1 on error



Executing Another Program from a Process (2)

- Parent uses **`fork()`** to create a child process
- Child uses **`exec()`** to replace its memory space with a new program
- Parent calls **`wait()`** to wait for child to terminate



execlp() Example

```
/*
   This program will

   1) Create a process
   2) Have the new process execute "ls -a -l“
      using execlp()
   3) Wait until the new process terminates

  */

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

```
int main()

{
        pid_t pid;

        /* fork a child process */
        pid = fork();
        if (pid < 0) { /* error occurred */
                printf("Fork Failed\n");
                exit(1);
        }
        else if (pid == 0) { /* child process */
                execlp("ls","ls","-a","-l",NULL);
                // still here?  Problem...
                printf("Exec error!\n");
                exit(1);
        }
        else { /* parent process */
                /* parent will wait for the child to complete */
                waitpid(pid, NULL, 0);
                printf("Child has terminated, so will I\n");
        }
        exit(0);
}
```

https://onlinegdb.com/BJ2QC18mD

execvp() Example

```
/*
   This program will

   1) Create a process
   2) Have the new process execute "ls -a -l“
      using execvp()
   3) Wait until the new process terminates

  */

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

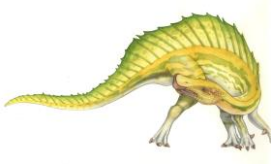
```
int main()
{
        pid_t pid;

        /* fork a child process */
        pid = fork();
        if (pid < 0) { /* error occurred */
                printf("Fork Failed\n");
                exit(1);
        }
        else if (pid == 0) { /* child process */
                char *args[4]={"ls", "-a", "-l", NULL};
                execvp("ls", args);
                // still here?  Problem...
                printf("Exec error!\n");
                exit(1);
        }
        else { /* parent process */
                /* parent will wait for the child to complete */
                waitpid(pid, NULL, 0);
                printf("Child has terminated, so will I\n");
        }
        exit(0);
}
```

https://onlinegdb.com/B10wAyL7D

File System Calls

- open(): open a file
- close(): close a file
- read(): read data from a file
- write(): write data to a file



Open()

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

int open(const char *pathname, int flags);

int open(const char *pathname, int flags, mode_t mode);

//open and possibly create a file

- Return value
  - On success, returns a file descriptor (a small, nonnegative integer) for use in subsequent system calls
  - On error, returns -1

An Open() Example

int fd = open(“file.txt”, O_WRONLY|O_CREAT|O_TRUNC, S_IRWXU|S_IROTH)

- **flags**
  - – O_WRONLY: write-only
  - – O_RDONLY: read-only
  - – O_RDWR: read/write
  - – O_CREAT: create the file if it does not exist
  - – O_TRUNC: if file already exists and access mode allows writing (i.e., O_RDWR or O_WRONLY), it will be truncated to length 0
- **mode** specifies the permissions to use in case a new file is created
  - – S_IRWXU: user (file owner) has read, write, and execute permission
  - – S_IROTH: others have read permission
  - – **mode** must be specified when O_CREAT is specified in flags, and is ignored otherwise.

Read(), Write(), Close()

#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count) // read from a file descriptor

• Reads up to count bytes from file descriptor fd into the buffer pointed to by buf

• Returns the number of bytes read on success (zero indicates end of file) and -1 on error

ssize_t write(int fd, const void *buf, size_t count) // write to a file descriptor

• Writes up to count bytes from the buffer pointed to by buf to file descriptor fd

• Returns the number of bytes written on success and -1 on error

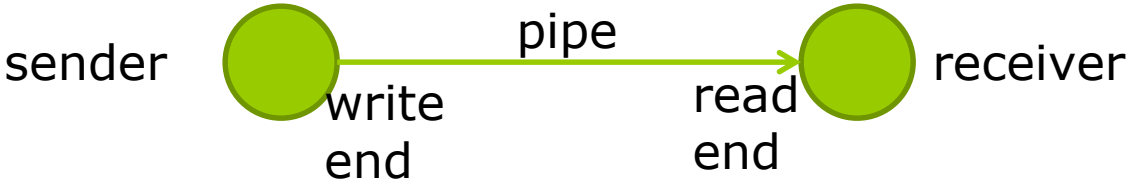
int close(int fd) //close a file descriptor

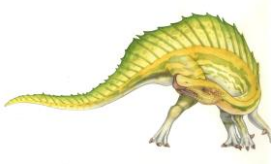
• Closes the file descriptor fd so that it no longer refers to any file and may be reused

• Returns 0 on success and -1 on error.

Pipes

- A pipe is a **unidirectional** data channel that can be used for interprocess communication
  - Sender puts data to one end (the **write-end** of the pipe)
  - Receiver gets data from the other end (the **read-end** of the pipe)





Creating Pipes

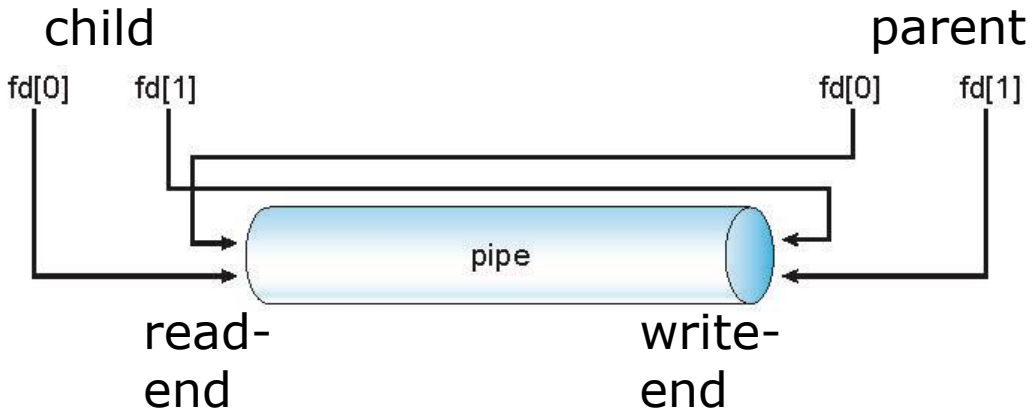
#include <unistd.h>

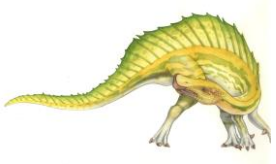
int pipe(int pipefd[2]) // create a pipe

- The array pipefd is used to return two file descriptors referring to the two ends of the pipe
  - fd[0] refers to the read-end of the pipe
  - fd[1] refers to the write-end of the pipe
  - Data written to the write end of the pipe is buffered by the kernel until it is read from the read end of the pipe.
- Returns 0 on success and -1 on error

Using Pipes (1)

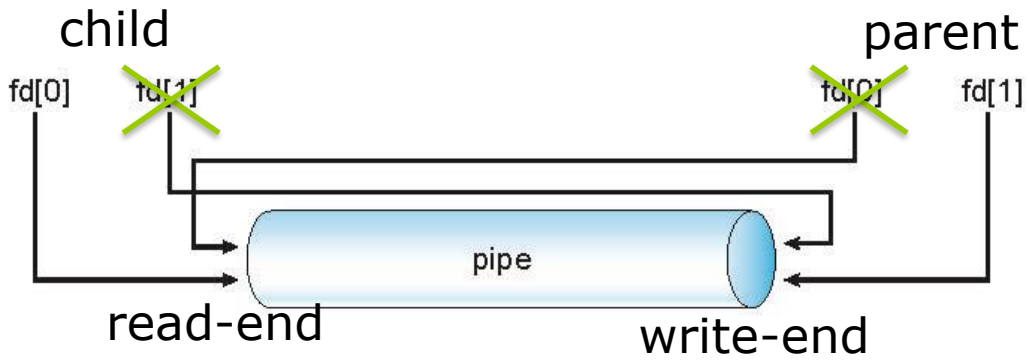
- Sending process and receiving process must have a common ancestor who creates a pipe
- A child process inherits open files from parent, and a pipe is a special type of file → Child inherits pipe from parent



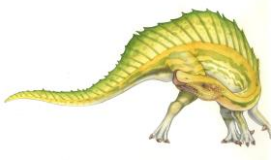


Using Pipes (2)

- Sending process calls write(fd[1], buf, len) to write to the pipe
- Receiving process calls read(fd[0], buf, len) to read from the pipe
- Each process should close the unused end of pipe to ensure that a process reading from the pipe can detect end-of-file when the writer has closed its end of the pipe
  - Sending process should close fd[0]
  - Receiving process should close fd[1]



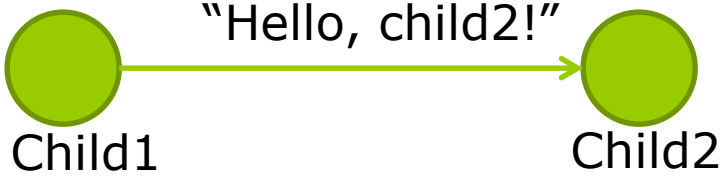
Parent sends message to child



Pipe Example

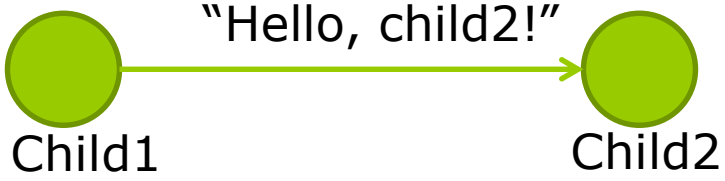
```
/*
 * This program creates two child processes
 * and a pipe. Child 1 sends a message to child2
 * via the pipe.
 */
#include <sys/wait.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>
void Child1(int fd)
// fd is the file descriptor to write the message to
{
char *message = "Hello, child2!";
int bytes = strlen(message)+1;
printf("Child1 sending %d byte message %s to
child2\n", bytes, message);
write(fd, message, bytes);
printf("Child1 exiting\n");
}
```

```
void Child2(int fd)
// fd is the file descriptor to read from
{
char buffer[256];
printf("Child2 waiting for message from
child1\n");
int sent = read(fd, buffer, 256);
printf("Child2 got %d byte message %s from
child1\n", sent, buffer);
printf("Child2 exiting\n");
}
```



Pipe Example (continued)

```
int main()
{
int pipefd[2];
if (pipe(pipefd) < 0) {
printf("Pipe error!\n");
return 1;
}
pid_t pid2 = fork();
if (pid2==0) {
// Child 2
close(pipefd[1]); //close write-end of pipe
Child2(pipefd[0]); //read from pipe
return 0;
}
```




```
pid_t pid1 = fork();
if (pid1==0) {
// Child 1
close(pipefd[0]); //close read-end of pipe
Child1(pipefd[1]); // write to pipe
return 0;
}
// Parent... close pipe and wait for children
close(pipefd[0]);
close(pipefd[1]);
waitpid(pid1, NULL, 0);
waitpid(pid2, NULL, 0);
printf("Everyone is done\n");
return 0;
}
```

https://onlinegdb.com/H16zyg87D

Redirection of Standard Input and Output

- In the UNIX shell, the symbols '<' and '>' cause input and output to be redirected
- For example
  - **ls -l > file1** sends the output of **ls -l** to file1
  - **head -5 < file1** reads from file1 instead of standard input
- We can use **dup2()** system call to redirect standard input and output

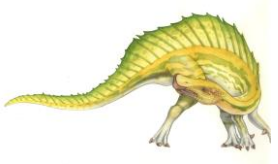
dup2()

#include <unistd.h>

int dup2(int oldfd, int newfd) //duplicate a file descriptor

- Makes newfd be a copy of oldfd
- Returns the new file descriptor, or -1 if an error occurred

- **Standard UNIX file descriptors**:
  0 = standard input (stdin)
  1 = standard output (stdout)
- **To redirect standard input:**
  int fd = open (“input.txt”, O_RDONLY);
  dup2(fd, 0); //whenever your program reads from stdin, it will read from fd
- **To redirect standard output:**
  int fd = open (“output.txt”, O_WRONLY|O_TRUNC);
  dup2(fd, 1); //whenever your program writes to stdout, it will write to fd



Pipeline Example

```
/* * This program:
* Creates two children processes and a pipe.
* Re-directs standard output of child1 to the pipe.
* Re-directs standard input of child2 to the pipe.
* Runs "yes no" in child1
* Runs "head -10" in child2
* Waits for all to finish
*/
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>
#include <string.h>
void Child1(int fd)
// fd is the file descriptor to write to
{
dup2(fd, 1);
close(fd);
execlp("yes", "yes", "no", NULL);
printf("Child 1 error!\n");
}
void Child2(int fd)
// fd is the file descriptor to read from
{
dup2(fd, 0);
close(fd);
execlp("head", "head", "-10", NULL);
printf("Child 2 error!\n");
}
int main()
{
int pipefd[2];
if (pipe(pipefd) < 0) {
printf("Pipe error!\n");
return 1;
}
int pid2 = fork();
if (0==pid2) { // Child 2
close(pipefd[1]); // I don't write to the pipe
Child2(pipefd[0]); // read end of pipe
return 0;
}
int pid1 = fork();
if (0==pid1) { // Child 1
close(pipefd[0]); // I don't read from the pipe
Child1(pipefd[1]); // write end of pipe
return 0;
}
// Parent... close pipe and wait for children
close(pipefd[0]); // if you take this out, we will hang!
close(pipefd[1]);
waitpid(pid1, NULL, 0);
waitpid(pid2, NULL, 0);
printf("Everyone is done\n");
return 0;
}
```

- We have gone through all the system calls you need to complete Project 1-UNIX Shell
- Start early!