- Definition Process
- UNIX process model
- process creation
- process fans and process chains
- process destruction and daemon processes
- inheritance and other process relationships
- implications of critical sections in concurrent processes

Process

A process is the basic active entity in most o.s models.

A process is basically a program in execution.

The execution of a process must progress in a sequential fashion.

In computing, a process is the instance of a computer program that is being executed.

It contains the program code and its activity.

Process Identification

- In UNIX each processes is identified by a unique integral value called the process ID (PID).
- Each process also has a parent process ID, which is initially the process ID of the process that created it.
- If this parent process terminates, the process is adopted by a system process so that the parent process ID always identifies a valid process.
  - getpid - > return the process ID
  - getppid -> returns parent process ID,

The pid_t is an unsigned integer type that represents a process ID.

SYNOPSIS

```
#include <unistd.h>
pid_t getpid(void);
pid_t getppid(void) ;
```

Note: Neither the getpid nor the getppid functions can return an error.

Process

Ex: Program to print process ID and its parent process ID using C.

```
#include <stdio.h>
#include <unistd.h>
int main (void) {
printf("I am process %ld\n", (long)getpid());
printf("My parent is %ld\n", (long)getppid());
return 0;
}
```

```
$ gcc -o pid1 pid.c
I am process 6786
My parent is 3908
```

Process

While creating the user's account, the System administrators assign

- ✓ unique integral user ID to each user and
- ✓ an integral group ID.

The user ID of most privileged *user, superuser or root*, is 0.
The root user is usually the system administrator.

A UNIX process has several user and group IDs that convey privileges to the process. These include the **real user ID, the real group ID, the effective user ID and the effective group ID.**

Usually, real and effective IDs are same but can be change by process.

Ex: a program runs with root privileges want to create a file on behalf of an ordinary user.
By setting the process's effective user ID to be that of this user, the process can create the files "as if" the user created them.

Note: Assume real and effective user and group IDs are the same.

Process

Functions to get group and user IDs of a process.

getegid(), geteuid(), getgid(), getuid(),

The gid_t and uid_t are integral types representing group and user IDs,

SYNOPSIS

```
#include <unistd.h>

effective IDs.
gid_t getegid(void);      // process ID
uid_t geteuid(void);      // group ID

real IDs.
git_t getgid(void);             // process ID
uid_t getuid(void);       // group ID
```

Note: functions can return an error.

Process

EX: prints out various user and group IDs for a process.

```
#include<stdio.h>
#include <unistd.h>
int main(void) {
printf("My real user ID is %5ld\n", (long)getuid());
printf("My effective user ID is %5ld\n", (long)geteuid());
printf("My real group ID is %5ld\n", (long)getgid());
printf("My effective group ID is %5ld\n", (long)getegid());
return 0;
}
```

```
$gcc -o pid2 pid2.c
My real user ID is  1000
My effective user ID is  1000
My real group ID is  1000
My effective group ID is  1000
```

??WAP to print the IDs of a parent and child process

Process State

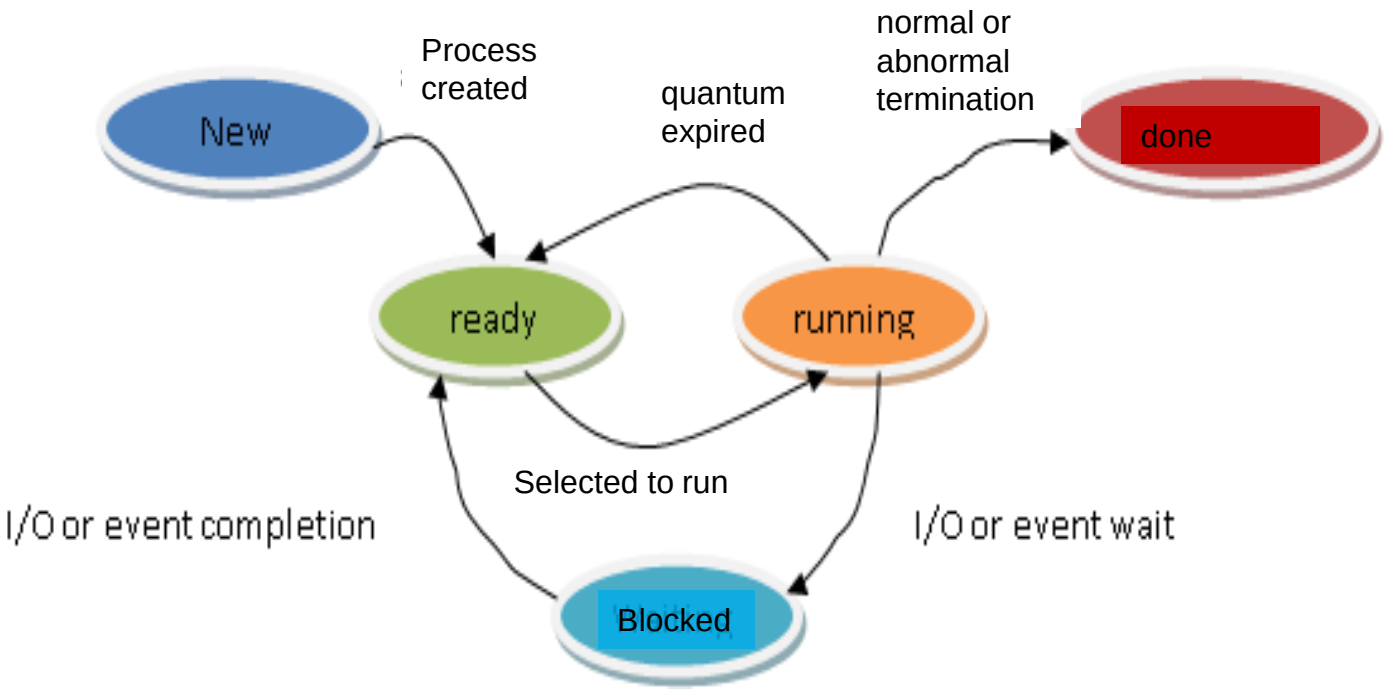
The status of a process at a particular time is referred as state of the process.

Common process states

	States	Meaning
1	new	being created
2	running	instructions are being executed
3	blocked	waiting for an event such as I/O
4	ready	waiting to be assigned to a processor
5	done	finished

Process

A state diagram is a graphical representation of the states of a process and the allowed transitions between states



Nodes: states
Edge: transitions

State diagram for a simple operating system

Process

- ✓ a program is undergoing the transformation into an active process, said to be in the new state.
- ✓ after transformation completes, the o.s puts the process in a queue of processes that are ready to run, ready state.
- ✓ a component of the o.s called the process scheduler selects a process to run, running state.
- ✓ A process in the blocked state is waiting for an event and is not eligible for execution. Also by ***sleep*** system command a process entered into this state.

Process

context switch is the act of removing one process from the running state and replacing it with another process.

operating systems needs about the process and its environment to restart it after a context switch s.t.

- process state,
- the status of program I/O,
- user and process identification,
- privileges,
- scheduling parameters,
- accounting information and
- memory management information, etc.

Process

To displays information about processes

ps utility:

SYNOPSIS

ps [-aA] [-G grouplist] [-o format]...[-p proclist] [-t termlist] [-U userlist]

default : displays information for processes associated with the user.

-a : displays information for processes associated with terminals.

-A : displays information for all processes.

-o : specifies the format of the output.

```
$ ps -a
PID TTY          TIME CMD
 8915 pts/2    00:00:00 ps

$ ps –la
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 R  1000  8931  3908  0  80   0 -  7467 -      pts/2    00:00:00 ps
```

Process

header	option	meaning
F	-l	flags (octal and additive) associated with the process
S	-l	process state
UID	-f, -l	user ID of the process owner
PID	(all)	process ID
PPID	-f, -l	parent process ID
C	-f, -l	processor utilization used for scheduling
PRI	-l	process priority
NI	-l	nice value
ADDR	-l	process memory address
SZ	-l	size in blocks of the process image
WCHAN	-l	event on which the process is waiting
TTY	(all)	controlling terminal
TIME	(all)	cumulative execution time
CMD	(all)	command name (arguments with -f option)

UNIX Process Creation : fork()

fork() : used to create a new process.

The calling process becomes the parent, and the created process is called the child.

The fork function copies the parent's memory image, so new process receives a copy of the address space of the parent.

Both processes continue at the instruction after the fork statement (executing in their respective memory images).

SYNOPSIS

```
#include <unistd.h>
pid_t fork(void);
```

returns

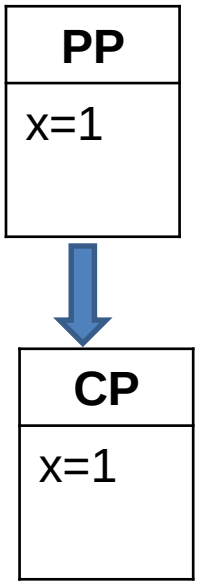
- =0 i.e 0 for the child
- >0 i.e child's process ID to the parent.
- <0 i.e returns –1, sets an errno and does not create a child errno = EAGAIN

(not having necessary resources for child or if limits on the number of processes would be exceeded)

Process

EX: both parent and child execute the x = 1

```
#include <stdio.h>
#include <unistd.h>
int main(void) {
int x;
x = 0;
fork();
x = 1;
printf("process %ld and x is %d\n",
(long)getpid(), x);
return 0;
}
```




```
$ ./fork1
process 9696 and x is 1
process 9697 and x is 1
```

Process

Ex: parent and child output their respective process IDs

```
#include <unistd.h>
#include <sys/types.h>
int main(void) {
pid_t childpid;
childpid = fork();
if (childpid == -1) {
perror("Failed to fork");
return 1;
}
if (childpid == 0) /* child code */
   printf("I am child %ld\n", (long)getpid());
else /* parent code */
   printf("I am parent %ld\n", (long)getpid());
return 0;
}
```

```
I am parent 9945
I am child 9946
```

note: It executes the second printf statement
output order is different

Process

```
//Check it
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main(void) {
pid_t childpid;
pid_t mypid;
mypid = getpid();
childpid = fork();
if (childpid == 0) /* child code */
   printf("I am child %ld, ID = %ld\n", (long)getpid(), (long)mypid);
else /* parent code */
   printf("I am parent %ld, ID = %ld\n", (long)getpid(), (long)mypid);
return 0;
}
```

The parent sets the mypid value to its process ID before the fork.
When fork , the child gets a copy of the process address space, including all variables.
Since the child does not reset mypid, the value of mypid for the child does not agree with the value returned by getpid.

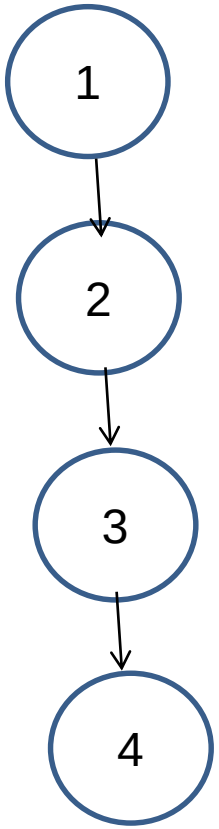
Process

Ex: A program that creates a chain of n processes, where n is a command-line argument

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main (int argc, char *argv[]) {
pid_t childpid = 0;
int i, n;
if (argc != 2){ /* check for valid number of command-line arguments */
fprintf(stderr, "Usage: %s processes only \n", argv[0]);
return 1;
}
n = atoi(argv[1]);
for (i = 1; i < n; i++)
  if (childpid = fork())
  break;
fprintf(stderr, "i:%d process ID:%ld parent ID:%ld child ID:%ld\n",i,
(long)getpid(), (long)getppid(), (long)childpid);
return 0;
}
```

```
$gcc -o fork3 fork3.c
$./fork3 3
i:1 process ID:11628 parent ID:3908 child ID:11629
i:2 process ID:11629 parent ID:1645 child ID:11630
i:3 process ID:11630 parent ID:11629 child ID:0
```

Process



??Check for large values of n.

Will the messages always come out ordered by increasing i

?? writes the messages to stdout, using printf, instead of to stderr, using fprintf?

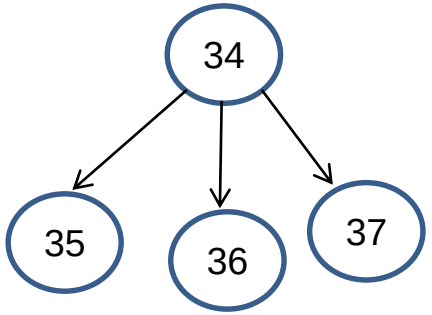
By default, the system buffers output written to stdout, so a particular message may not appear immediately after the printf returns.

Messages to stderr are not buffered, written immediately. For this reason, use stderr for debugging messages.

Process

Ex: creates a fan of n processes where n is passed as a command-line argument.

```
int main (int argc, char *argv[]) {
pid_t childpid = 0;
int i, n;
if (argc != 2){ /* check for valid number of command-line arguments */
   fprintf(stderr, "Usage: %s processes only\n", argv[0]);
return 1;
}
n = atoi(argv[1]);
for (i = 1; i < n; i++)
  if (childpid = fork()<=0)
  break;
fprintf(stderr, "i:%d process ID:%ld parent ID:%ld child ID:%ld\n",i, (long)getpid(), (long)getppid(),
(long)childpid);
return 0;
}
```



Generated process fan

```
./fork 4
i:4 process ID:9634 parent ID:4309 child ID:0
i:1 process ID:9635 parent ID:9634 child ID:1
i:2 process ID:9636 parent ID:9634 child ID:1
i:3 process ID:9637 parent ID:9634 child ID:1
```

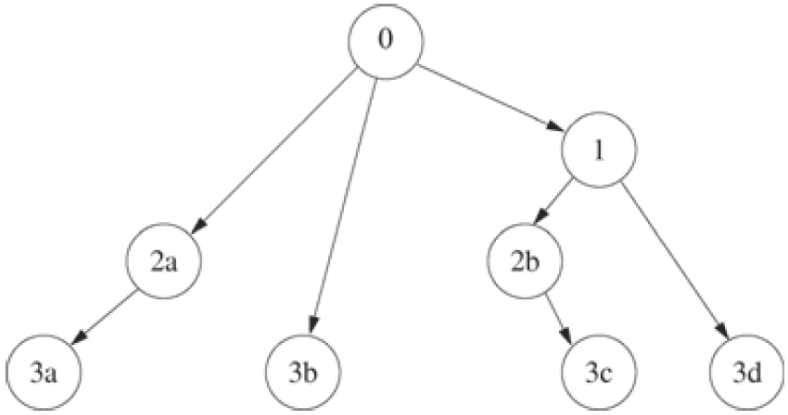
Process

?? replace the in the program

- (childpid = fork()) <= 0 with (childpid = fork()) == -1

In this case, all the processes remain in the loop unless the fork fails.
Each iteration of the loop doubles the number of processes, forming a tree n is 4.

?? Represent as a figure



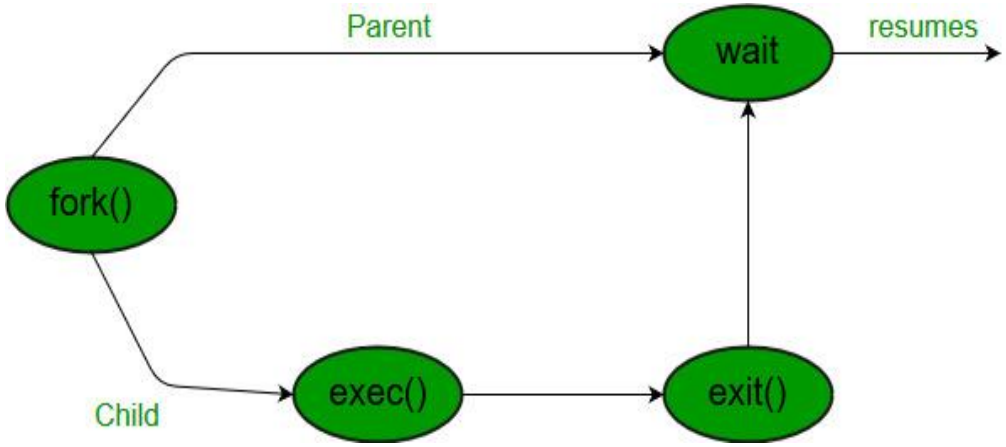
each process by a circle labeled with the i value at the time it was created.
The original process has a 0 label.
The lowercase letters distinguish processes that were created with the same value of i.

Process

wait() function

process creates a child, both parent and child proceed with execution from fork().

The parent can execute wait or waitpid to block until the child finishes.



The wait function causes the caller to suspend execution until a child's status becomes available or until the caller receives a signal.

SYNOPSIS

```
#include <sys/wait.h>
pid_t wait(int *stat_loc);
```

Process

waitpid() function allows

- a parent to wait for a particular child.
- a parent to check whether a child has terminated without blocking

Read three parameters:

1. a pid,
   - pid = –1 (waits for any child)
   - pid >0 (waits for the specific child whose process ID is pid)
   - pid =0 (waits for any child in the same process group as the caller)
   - pid < –1 (waits for any child in the process group specified by the absolute value of pid
2. a pointer to a location for returning the status
3. a flag specifying options.

Process

SYNOPSIS

```
#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

options parameter : is the bitwise inclusive OR of one or more flags

- WNOHANG : return even if the status of a child is not immediately available.
- WUNTRACED : the status of unreported child processes that have stopped.
- If an error occurs, these functions return –1 and set errno

errno	**cause**
ECHILD	caller has no unwaited-for children (wait), or process or process group specified by pid does not exist (waitpid), or process group specified by pid does not have a member that is a child of caller (waitpid)
EINTR	function was interrupted by a signal
EINVAL	options parameter of waitpid was invalid

Process

EX: code segment waits for a child

```
#include <stdio.h>
#include <unistd.h>
#include<sys/wait.h>
#include<stdlib.h>
int main(void) {
pid_t childpid;
fork();
childpid = wait(NULL);
if (childpid != -1)
  printf("Waited for child with pid %d\n", childpid);
return 0;
}
```

o/p

```
$ ./a.out
Waited for child with pid 10084
```

Process

EX: code segment waits for a child

```
int main()
{
    if (fork()== 0)
        printf("I am child process\n");
    else
    {   printf("I am parent process\n");
        wait(NULL);
        printf("child has terminated\n");
    }

return 0;
}
```

o/p:

```
I am parent process        --- 1
I am child process         --- 2
child has terminated       ---3
```

OR

```
I am child process
I am parent process
child has terminated
```

Note: Never the line 3 comes before line 1,2

Process

EX: code segment waits for a child

```
#include<stdio.h>
#include<unistd.h>
#include<sys/wait.h>
#include<errno.h>
int main()
{
pid_t childpid;
fork();
printf("PID=%d\n",getpid());
printf("PPID=%d\n",getppid());
while( childpid = waitpid(-1,NULL,WNOHANG))
  if ((childpid == -1) && (errno!=EINTR))
            break;
printf("Waited for pid %ld\n", (long)childpid);
 return 0;
}
```

o/p:

```
PID=8324
PPID=3079
Waited for pid 0

PID=8325
PPID=8324
Waited for pid -1
```

Note: waits for all children that have finished but avoids blocking if there are no children whose status is available

Process

??What happens when a process terminates, but its parent does not wait for it?

A process which has finished its execution but still has entry in the process table to report to its parent process is known as a zombie process.

A child process first becomes a zombie before removed from the process table.

```
#include<stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main() {
    pid_t pid = fork();
   // Parent process
    if (pid > 0)
       { sleep(5);
         printf("Hello");}
   // Child process
    else
        exit(0);
    return 0;
}
```

Process

??What happens when a process terminates, but its parent does not wait for it?

A process whose parent process no more exists i.e. either finished or terminated without waiting for its child process to terminate is called an orphan process.

```
int main(){
    int pid = fork();
    if (pid > 0)
        printf("in parent process\n");
 // Note that pid is 0 in child process
 // and negative if fork() fails
    else if (pid == 0)    {
        sleep(10);
        printf("in child process\n");
    }
    return 0;
}
```

```
$ ./a.out
in parent process
MN$ in child process
```

Note: After 10 sec child invoked automatically

Orphan process is adopted by a special system process, called init and has process ID equal to 1.

Process

Refer Example 3.15 --- 3.21

Process

Status values : stat_loc argument of wait or waitpid

It is a pointer to an integer variable

If it is not NULL, then store the return status of the child in this location.

The child returns its status by calling exit, _exit, _Exit or return from main.

0 value - EXIT_SUCCESS;

any other value -- EXIT_FAILURE.

The parent can only access the 8 least significant bits of the child's return status.

Process

SYNOPSIS

#include <sys/wait.h>

WIFEXITED(int stat_val)
WEXITSTATUS(int stat_val)
WIFSIGNALED(int stat_val)
WTERMSIG(int stat_val)
WIFSTOPPED(int stat_val)
WSTOPSIG(int stat_val)

Used in pairs

WIFEXITED evaluates to a nonzero value when the child terminates normally.
Then WEXITSTATUS evaluates to the low-order 8 bits returned by the child through _exit(), exit() or return from main.

WIFSIGNALED is nonzero value when the child terminates because of uncaught signal.
then WTERMSIG evaluates to the number of the signal that caused the termination.

The WIFSTOPPED is nonzero value if a child is currently stopped.
then WSTOPSIG evaluates number of the signal that caused the child process to stop.

Process

Program to determines the exit status of a child

Refer Example-3.22

(include header file “"restart.h”)

Process Termination

Process termination can be normal or abnormal

The activities performed during process termination include

- cancelling pending timers and signals,
- releasing virtual memory resources,
- releasing other process-held system resources such as locks, and closing files that are open.
- o.s record process status and used resources, notify its parent process

Process Termination

In UNIX, a process does not completely release its resources after termination until the parent waits for it.

If its parent is not waiting when the process terminates, the process becomes a zombie.

A zombie is an inactive process whose resources are deleted later when its parent waits for it.

When a process terminates, its orphaned children and zombies are adopted by a special system process.

Process Termination

A normal termination occurs under the following conditions.

- return from main
- Implicit return from main (the main function falls off the end)
- Call to exit, _Exit or _exit

In C exit function

1. calls user-defined exit handlers that were registered in the reverse order of registration.
2. flushes any open streams that have unwritten buffered data
3. closes all open streams.
4. exit removes all temporary files that were created by tmpfile()
5. terminates control

- return statement from main has the same effect as calling exit
- _Exit and _exit functions do not call user-defined exit handlers

Process Termination

SYNOPSIS

#include <stdlib.h> (ISO C)

void exit(int status);

void _Exit(int status);

SYNOPSIS (POSIX)

#include <unistd.h>

void _exit(int status);

Input arg: small integer parameter, status

Ex: exit() function in c

```
#include <stdio.h>
#include <stdlib.h>
int main () {
   printf("Start of the program....\n");
   printf("Exiting the program....\n");
   exit(0);
   printf("End of the program....\n");
   return(0);
}
```

```
$ ./a.out
Start of the program....
Exiting the program....
```

Process Termination

C atexit (C library function) installs a user-defined exit handler, specified function func to be called when the program terminates

- Exit handlers are executed on a last-installed-first-executed (reverse) order when the program returns from main or calls exit.
- use multiple calls to atexit to install several handlers.
- takes a single parameter, to be executed as a handler.

SYNOPSIS

```
#include <stdlib.h>
int atexit(void (*func)(void));
```

.

Process Termination

Ex: atexit() in C

```
#include <stdio.h>
#include <stdlib.h>
void functionA () {
   printf("This is functionA\n");
}
int main () {
   /* register the termination function */
   atexit(functionA );
   printf("Starting  main program...\n");
   printf("Exiting main program...\n");
   return(0);
}
```

```
$ ./a.out
Starting  main program...
Exiting main program...
This is functionA
```

Process Termination

A program showtime.c with an exit handler that outputs CPU usage.

Display statistics about the time used by the program and its children to be output to standard error before the program terminates.

Use times function returns timing information in the form of the number of clock ticks.
The showtimes function converts the time to seconds by dividing by the number of clock ticks per second (found by calling sysconf).

Open program showtime.c

Process Termination