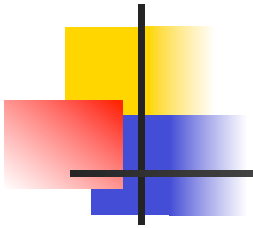
Sample Program

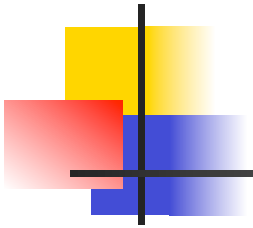


```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>

main()
{
        int forkresult, statvar, waitedfor;

        printf ("%d: I am the parent.\n", getpid ());
        printf ("%d: I am going to fork...\n", getpid());

        if ( (forkresult=fork()) != 0)  /* parent */

                printf ("%d: My child's pid is %d\n", getpid(), forkresult);

        else   /* child */

                printf ("%d: Hi! I am the child.\n", getpid());

        printf ("%d: So this whom?\n", getpid());
}
```

Sample Execution

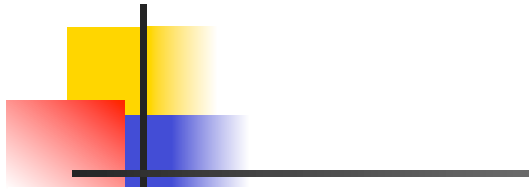
```
 $./forkexec
3750: I am the parent.
3750: I am going to fork...
3750: My child's pid is 3751
3750: So this whom?
3751: Hi! I am the child.
3751: So this whom?
$./forkexec
3752: I am the parent.
3752: I am going to fork...
3752: My child's pid is 3753
3752: So this whom?
3753: Hi! I am the child.
3753: So this whom?
$./forkexec
3754: I am the parent.
3754: I am going to fork...
3754: My child's pid is 3755
3755: Hi! I am the child.
3754: So this whom?
3755: So this whom?
```

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

main()
{
        int forkresult;

        printf ("%d: I am the parent.\n", getpid ());
        printf ("%d: I am going to fork...\n", getpid());

        forkresult = fork();
        printf ("%d: And the forkresult is %d\n", getpid(), forkresult);
        if (forkresult < 0)         /* error */

                printf ("ERROR: Fork failed \n");

        else if ( forkresult > 0)  /* parent */

                printf ("%d: My child's pid is %d\n", getpid(),
forkresult);

        else {                      /* child */

                printf ("%d: Hi! I am the child.\n", getpid());

        }
        printf ("%d: So this whom?\n", getpid());

        exit(1);
}
```



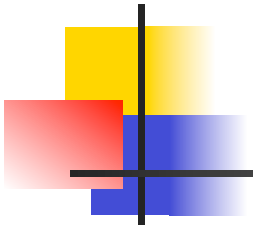
```
$./forkexec
6440: I am the parent.
6440: I am going to fork...
6440: And the forkresult is 6441
6441: And the forkresult is 0
6440: My child's pid is 6441
6441: Hi! I am the child.
6441: So this whom?
6440: So this whom?
$./forkexec
6443: I am the parent.
6443: I am going to fork...
6443: And the forkresult is 6444
6444: And the forkresult is 0
6443: My child's pid is 6444
6443: So this whom?
6444: Hi! I am the child.
6444: So this whom?
```

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

main()
{
        int forkresult, statvar, waitedfor;

        printf ("%d: I am the parent.\n", getpid ());
        printf ("%d: I am going to fork...\n", getpid());

        if ( (forkresult=fork()) != 0) { /* parent */

                printf ("%d: My child's pid is %d\n", getpid(), forkresult);

         } else {  /* child */

                printf ("%d: Hi! I am the child.\n", getpid());

                printf ("%d: Trying ls now...\n", getpid());
                execlp ("ls", "ls", NULL);
                printf ("%d: ls completed\n", getpid());

        }

        printf ("%d: So this whom?\n", getpid());
        exit(1);
}
```



```
$./forkexec
3880: I am the parent.
3880: I am going to fork...
3880: My child's pid is 3881
3880: So this whom?
3881: Hi! I am the child.
3881: Trying ls now...
Assignment1.pdf      ProgExecExample.pdf  forkexec1.c
Lecture3.pdf         ProgExecExample.ppt  forkexec2.c
Lecture3.ppt         Syllabus.html        index.html
Lecture4.pdf         Syllabus.pdf
Lecture4.ppt         forkexec

$ ./forkexec
3882: I am the parent.
3882: I am going to fork...
3882: My child's pid is 3883
3883: Hi! I am the child.
3882: So this whom?
3883: Trying ls now...
$ Assignment1.pdf      ProgExecExample.pdf  forkexec1.c
Lecture3.pdf         ProgExecExample.ppt  forkexec2.c
Lecture3.ppt         Syllabus.html        index.html
Lecture4.pdf         Syllabus.pdf
Lecture4.ppt         forkexec
```

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

main()
{
        int forkresult, statvar, waitedfor;

        printf ("%d: I am the parent.\n", getpid ());
        printf ("%d: I am going to fork...\n", getpid());

        if ( (forkresult=fork()) != 0) { /* parent */

                printf ("%d: My child's pid is %d\n", getpid(), forkresult);
                wait (NULL);
                printf ("%d: So this whom?\n", getpid());

        } else {  /* child */

                printf ("%d: Hi! I am the child.\n", getpid());

                printf ("%d: Trying ls now...\n", getpid());
                execlp ("ls", "ls", NULL);
                printf ("%d: ls completed\n", getpid());

        }
        exit(1);
}
```



```
$./forkexec
6462: I am the parent.
6462: I am going to fork...
6462: My child's pid is 6463
6463: Hi! I am the child.
6463: Trying ls now...
Assignment1.pdf          ProgExecExample.ppt      forkexec11.c
Lecture3.pdf             Syllabus.html            forkexec2.c
Lecture3.ppt             Syllabus.pdf             forkexec21.c
Lecture4.pdf             UNIXForkExecExample.pdf  index.html
Lecture4.ppt             forkexec
ProgExecExample.pdf      forkexec1.c
6462: So this whom?
```

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 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 State Transition Diagram

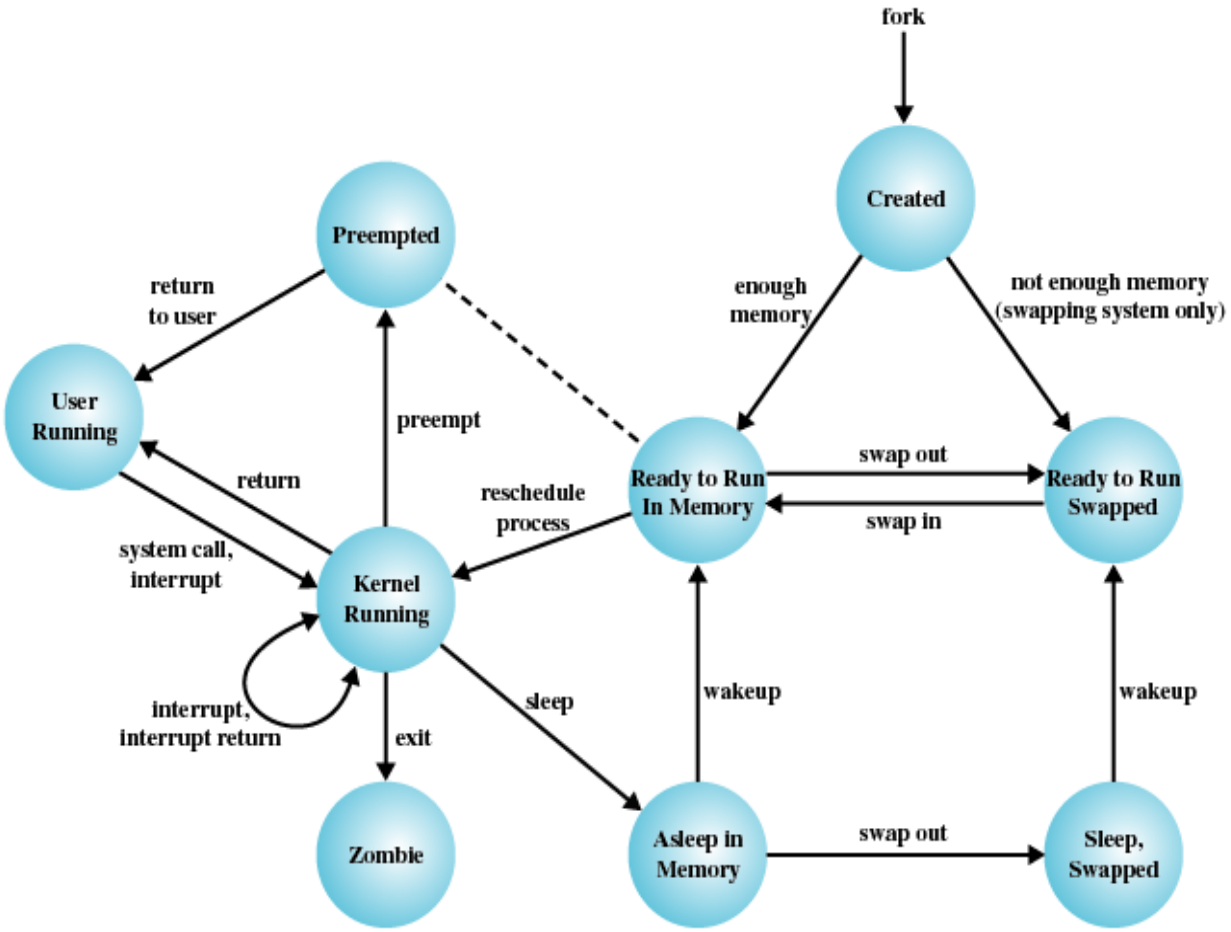


Figure 3.17 UNIX Process State Transition Diagram