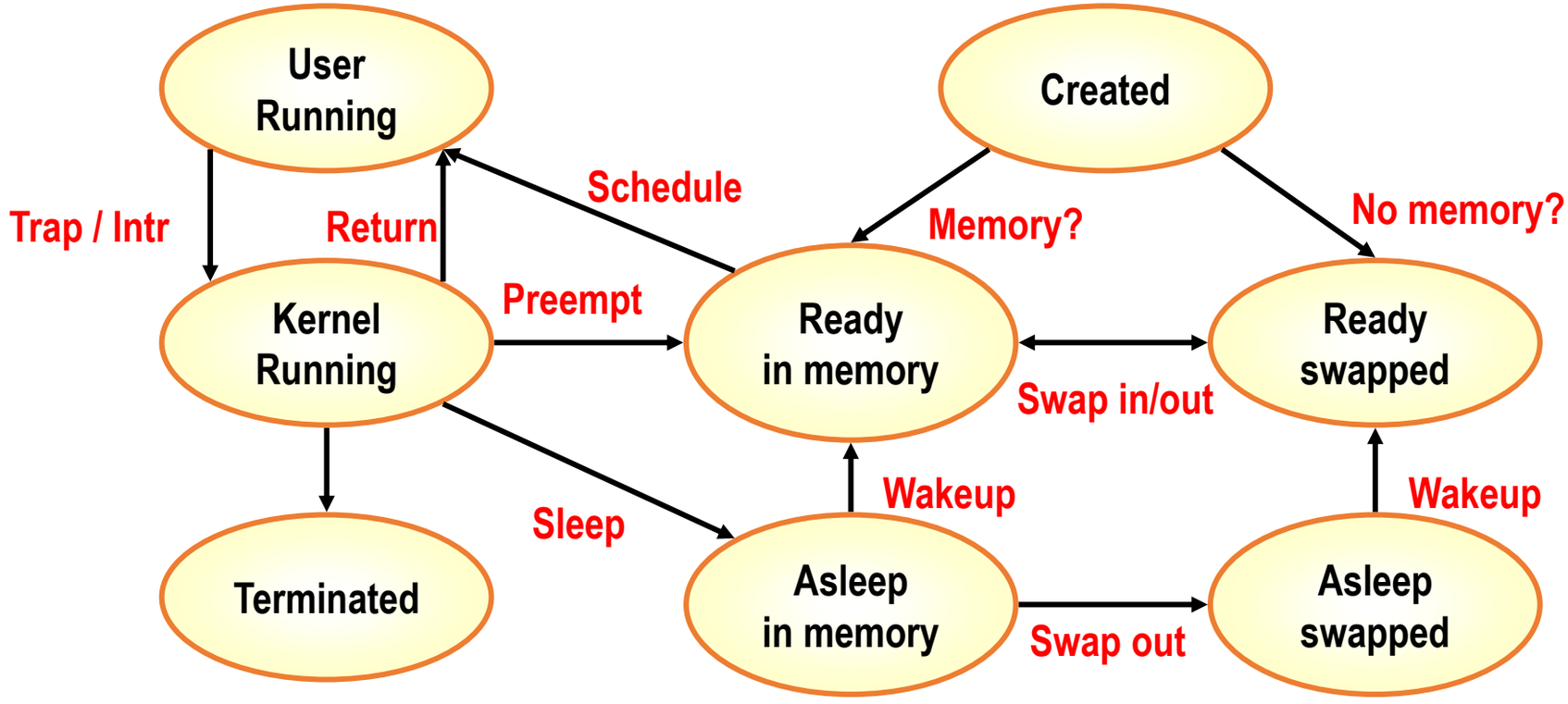
Process

- A process is a program in execution
- Contents
  - Process control block
    - Process identification
    - Process state information
    - Process control information
  - User stack
  - Private user address space (program, data)
  - Shared address space

Process State Transitions



How to create a new process?

- The fork() system call
  - It creates a new process as a *child process* of the calling process (*parent*)
  - Both have similar code segments
  - The child gets a copy of the parents data segment at the time of forking

- How can the child realize that it is the child and not the parent?
- How can we make the child and parent do different things?

The return value of fork()

fork( ) returns a value to both parent and child

- The parent receives the process id of the child
- The child receives 0 (zero)

Key idea:

```
if (fork() == 0)
        { /* I am the child process */ }
else
        { /* I am the parent process */ }
```

The first program: fork1.c

```
#include <stdio.h>
#include <sys/ipc.h>
main( )
{
   if (fork( ) == 0) {  /* Child */
            while (1) {          for (i=0; i<100000; i++) ;
                                 printf("\t\t\t Child executing\n ");        }
   }
   else {               /* Parent */
            while (1) {          for (i=0; i<100000; i++) ;
                                 printf("Parent executing\n");  }
   }
}
```

Waiting for child termination

- The parent process can wait for the child process to terminate using the call:

  waitpid(pid, NULL, 0)

  -- where pid is the identifier of the child process (returned by fork( ))
  -- what are the other two parameters?

The second program: fork2.c

```
#include <stdio.h>
#include <sys/ipc.h>
main()
{
   int i, x = 10, pid1, pid2 ;
   printf("Before forking, the value of x is %d\n", x);

   if ( ( pid1 = fork( ) ) == 0) {  /* First child process */
            for (i=0 ; i < 5; i++) {
               printf("\t\t\t At first child: x= %d\n", x);
               x= x+10;
               sleep(1) ; /* Sleep for 1 second */
            }
   }
```

```
else {      /* Parent process */

   if ( ( pid2 = fork( ) ) == 0) {  /* Second child */
            for (i=0 ; i < 5; i++) {
               printf("\t\t\t\t\t At second child: x= %d\n", x);
               x= x+20;  sleep(1) ; /* Sleep for 1 second */
            }
   }
   else {   /* Parent process */
            waitpid(pid1, NULL, 0);
            waitpid(pid2, NULL, 0);
            printf("Both children terminated\n");
   }
}}
```

Points to ponder: fork3.c

```
#include <stdio.h>
#include <sys/ipc.h>
main( )
{
   int x=0, pid;
   printf("Hello!");

   if ( ( pid = fork() ) == 0) {   /* Child */
     printf("\nChild:\t Address of x: %x\t
                        Value of x: %d \n", &x, x);
     x = 20;
     printf("Child:\t Address of x: %x\t
                        Value of x: %d \n", &x, x);
   }
   else {   /* Parent */
     waitpid(pid, NULL, 0);
     printf("\nParent:\t Address of x: %x\t
                        Value of x: %d \n", &x, x);
     x = 10;
     printf("Parent:\t Address of x: %x\t
                        Value of x: %d \n", &x, x);

   }
}
```

- How many times is Hello! printed?
- Is the address of x printed by the parent and child the same, or different?

EXEC, PIPE, DUP

Exec

- System calls that allow a process to execute a specified program
  - Process identifier remains the same.
  - There is no return from exec.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/ipc.h>
main()
{
   execlp("cal","cal","2001",NULL);
   printf("This statement is not executed if execlp succeeds.\n");
}
```

Pipe

- The pipe( ) system call
  - Creates a pipe that can be shared between processes
  - It returns two file descriptors,
    - One for reading from the pipe, the other for writing into the pipe

```
#include <stdio.h>
#include <unistd.h>   /* Include this file to use pipes */
#define BUFSIZE 80
main()
{
   int fd[2], n=0, i;
   char line[BUFSIZE];

   pipe(fd);  /* fd[0] is for reading, fd[1] is for writing */
```

```
   if (fork() == 0) {
     close(fd[0]);      /* The child will not read */
     for (i=0; i < 10; i++) {
            sprintf(line,"%d",n);
            write(fd[1], line, BUFSIZE);
            printf("Child writes: %d\n",n);  n++; sleep(2);
   }}
   else {
     close(fd[1]); /* The parent will not write */
     for (i=0; i < 10; i++) {
            read(fd[0], line, BUFSIZE);
            sscanf(line,"%d",&n);
            printf("\t\t\t Parent reads: %d\n",n);
}  }}
```

Dup

- The dup(fd) system call:
  - Copies the descriptor, fd, into the first empty slot in the file descriptor table of the process
  - Recall that the 0th location of the FD table is for stdin and the 1st location of the FD table is for stdout.
  - We can use this information to use close( ) and dup( ) for redirecting stdin and/or stdout.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/ipc.h>
main( )
{
   int fd[2], n=0, i;
   pipe(fd);
```

```
   if (fork() == 0) {   /* Child process */
            close(1) ; dup(fd[1]) ; /* Redirect the stdout of this process to the pipe. */
            close(fd[0]);
            for (i=0; i < 10; i++) { printf("%d\n",n);  n++; }
   }
   else {                  /* Parent process */
            close(0) ; dup(fd[0]) ; /* Redirect the stdin of this process to the pipe */
            close(fd[1]);
            for (i=0; i < 10; i++) {  scanf("%d",&n);  printf("n = %d\n",n); sleep(1);  }
} }
```