A Tutorial on Unix System Calls

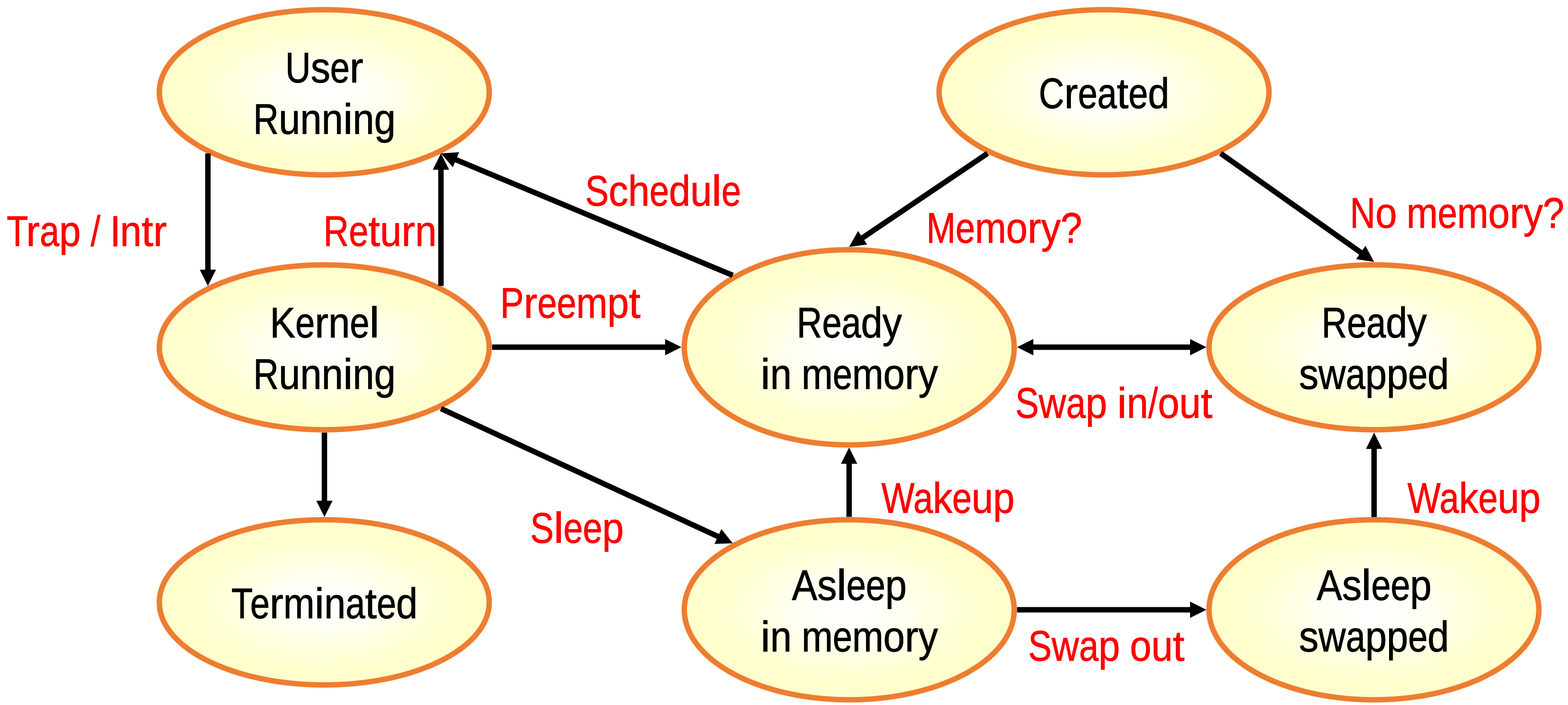
Inter-Process Communication (IPC)

PROF. PALLAB DASGUPTA
INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR
pallab@cse.iitkgp.ernet.in
http://cse.iitkgp.ac.in/~pallab

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\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);  }
} }
```

SHARED MEMORY

Shared Memory System Calls

- Creation:

  int shmid = shmget(IPC_PRIVATE, <no of bytes>, 0777|IPC_CREAT)

  - This call creates the shared memory segment and returns its identifier

- Attach:

  char * shmat(shmid, 0, 0)

  - This call attaches the shared memory segment with the logical address space of the calling process and returns the logical address

Using shared memory: shm.c

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
main()
{
    int shmid, *a, *b, i;

    /* Acquire a shared array of 2 integers */
    shmid = shmget( IPC_PRIVATE,
                    2*sizeof(int), 0777|IPC_CREAT);
```

```
    if ((pid = fork()) == 0) {          /* Child */
            b = (int *) shmat( shmid, 0, 0 );   /* Attach to child */
            for( i=0; i< 10; i++) {
                    sleep(1);
                    printf("\t\t\t Child reads: %d,%d\n",b[0],b[1]);
    }       }
    else {                  /* Parent */
            a = (int *) shmat( shmid, 0, 0 );  /* Attach to parent */
            a[0] = 0; a[1] = 1;
            for( i=0; i< 10; i++) {
                    sleep(1); a[0] = a[0] + a[1]; a[1] = a[0] + a[1];
                    printf("Parent writes: %d,%d\n",a[0],a[1]);
            }
            waitpid( pid );
}}
```

SEMAPHORES

Semaphores

- Semaphores are system variables used for process synchronization. You may think of a semaphore, s, as a variable maintained by the system.
  - A semaphore can be obtained by a semget() system call.
  - Its initial value can be set by the semctl() system call.

- There are two common operations that a process can perform on a semaphore, s, namely:
  - P(s) or wait(s): If the value of s is greater than 0, then this operation decrements the value of s and the calling process continues. Otherwise, if s is 0, then the calling process is blocked on s.
  - V(s) or signal(s) : If any process is blocked on s, then this unblocks (wakes up) the earliest among the processes blocked on s. Otherwise, the value of the semaphore is incremented.
  - In UNIX/Linux, both P(s) and V(s) can be done with the semop() system call with appropriate parameters.

Initialization

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>          /* Include this to use semaphores */

/* We will define the P(s) and V(s) operations in terms of the semop() system call. The syntax of semop is as follows:

        int semop ( int semid, struct sembuf *sops, unsigned nsops)

where semid is the semaphore identifier returned by the semget() system call. The second parameter is a pointer to a structure which we must pass. The fields of this structure indicates whether we wish to perform a P(s) operation or a V(s) operation. Refer to the system manual for the third parameter -- we will always use 1 for this parameter. */

#define P(s) semop(s, &pop, 1)  /* pop is the structure we pass for doing the P(s) operation */
#define V(s) semop(s, &vop, 1)  /* vop is the structure we pass for doing the V(s) operation */
```

Initialization

```
main()
{
   int shmid;
   int *a, *b, i, j ;

   int semid1, semid2 ;
   struct sembuf pop, vop ;

/* In the following  system calls, observe the similarity in the first and
third parameters. In a previous example (shm.c) we had studied these
parameters for shmget() – they have the same meaning for semget().
The second parameter indicates the number of semaphores under
this semid. Throughout this lab, give this parameter as 1. If we
require more semaphores, we will take them under different semids
through separate semget() calls. */

   shmid = shmget(IPC_PRIVATE, 2*sizeof(int), 0777|IPC_CREAT);
   semid1 = semget(IPC_PRIVATE, 1, 0777|IPC_CREAT);
   semid2 = semget(IPC_PRIVATE, 1, 0777|IPC_CREAT);
```

```
/* The following system calls sets the values of the
semaphores semid1 and semid2 to 0 and 1 respectively. */

      semctl(semid1, 0, SETVAL, 0);
      semctl(semid2, 0, SETVAL, 1);

/*  We now initialize the sembufs pop and vop so that pop
is used for P(semid) and vop is used for V(semid). For the
fields sem_num and sem_flg refer to the system manual.
The third field, namely sem_op indicates the value which
should be added to the semaphore when the semop()
system call is made. Going by the semantics of the P and
V operations, we see that pop.sem_op should be -1 and
vop.sem_op should be 1. */

      pop.sem_num = vop.sem_num = 0;
      pop.sem_flg = vop.sem_flg = SEM_UNDO;
      pop.sem_op = -1 ; vop.sem_op = 1 ;
```

Use of Semaphores

/* We now illustrate a producer-consumer situation.

- The parent process acts as the producer and the child process acts as the consumer.
- Initially semid1 is zero, hence the consumer blocks.
- Since semid2 is one, the producer produces (in this case writes some values into the shared memory). After this it wakes up the consumer through the V(semid1) call.
- The consumer reads the value and in turn performs V(semid2) to wake up the producer.

Trace through the code and work out the values of the two semaphores and see how they synchronize the producer and the consumer to wait for each other. */

```
if (fork() == 0) {
        /* Child Process::  Consumer */
        b = (int *) shmat(shmid, 0, 0);
        while (1) {  P(semid1);
                printf("\t\t\t\t Consumer reads %d, %d\n",b[0],b[1]);
                V(semid2);  }
}
else {
        /* Parent Process:: Producer */
        a = (int *) shmat(shmid, 0, 0);
        a[0] = 0; a[1] = 1;
        while (1) { sleep(1);
                P(semid2);
                a[0] = a[1] + 1;
                a[1] = a[0] + 1;
                printf("Producer writes %d, %d\n", a[0], a[1]);
                V(semid1); }
}
```