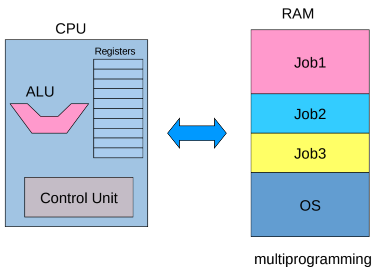
Preview

- Process Control
  - What is process?
  - Process identifier
  - The fork() System Call
  - File Sharing
  - Race Condition

What is a Process

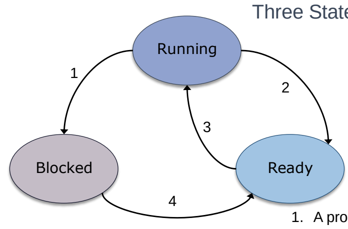
- A key concept in OS is the process
- Process – a program in execution
- Once a process is created, OS not only reserve space (in Memory) for the process but also need spaces (process table, page table …)to keep tracking the process.
- Process associated with
  - **Address space** – where the executable program, program data, stack and heap are allocated in a memory
  - **Set of registers** (Program counter, stack pointer, and other registers)
  - All other information for executing the process.

What is a Process



What is a Process

Three States for a Process

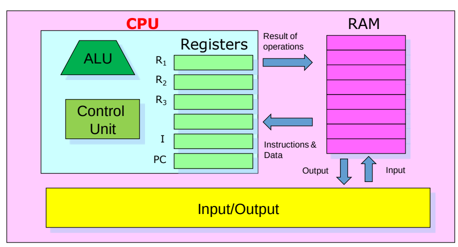


1. A process blocked for I/O
2. A process time out
3. A process scheduled
4. A process finish it's I/O

A Computer System
(Computer Structure: Von Newmann Bottleneck)

- In the von Neumann architecture, programs and data are held in memory; the processor and memory are separate and data moves between the two.
- The von Neumann bottleneck (memory stall) is a limitation on throughput caused by the standard personal computer architecture.
  - Throughput is a measure of how many units of information a system can process in a given amount of time.
- Since processor calculation speeds are much faster than data movement between memory and CPU, it cause bottleneck!

A Computer System
(Computer Structure: Von Newmann)

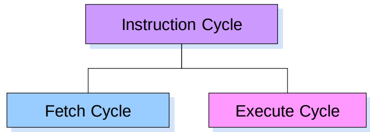


A Computer System
(Computer Structure: Von Newmann)

- John von Neumann was a Hungarian-American mathematician, physicist, computer scientist, and polymath.
- He made major contributions to a number of fields, including mathematics, physics, economics, computing, and statistics.
- Born: December 28, 1903, Budapest, Hungary
- Died: February 8, 1957,

Process Instruction Cycle

•The microprocessor's main task is to execute instructions.
•The ***instruction cycle*** is therefore at the heart of understanding the function and operation of the microprocessor.

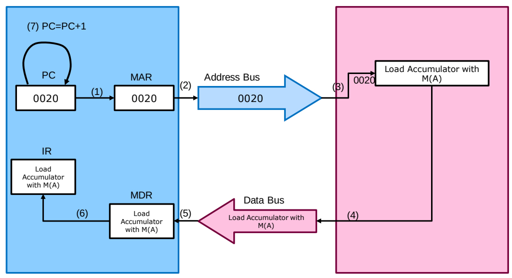


Process Instruction Cycle

Fetch cycle

1. Reading the address of the instruction in (PC) to be executed from the memory and
2. Loading it into the Instruction register (IR).
3. Program Counter register (PC) is modified to point at the next valid instruction.

Process Instruction Cycle



Process Instruction Cycle

Execute cycle

- The contents of the IR are decoded and executed.
- The execution may result in a variety of actions depending on the type of instruction.
- It may be a self contained instruction, or it can involve interaction with memory and ALU.

What is a Process

- All information about each process is stored in an operating system table called process table (**process control block**).
- If a process is suspended (ready or block state), information for the snapshot of the process are stored in its process table.
- Once the process resume a CPU time, all information for the process execution are copy back from its process table

Process Identifiers

- When a process is created, kernel provides unique process ID, a non-negative integer.
- When a process terminate, its ID becomes reusable for a newly created process.
- There are couple of process ID numbers which is used by system itself.
  - Process ID 0: a process scheduler (CPU scheduler)
  - Process ID 1: the systemd in LINUX (init in UNIX)process

Process Identifiers

- The ps command show the process we are running, the another user is running, or all the process on the system.
  - To see every process on the system using standard syntax:
    - ps -e
    - ps -ef
    - ps -eF
    - ps -ely
  - To see every process on the system using BSD syntax:
    - ps ax
    - ps axu
  - To print a process tree:
    - ps -ejH
    - ps axjf
  - To get security info:
    - ps axZ
    - ps -eM

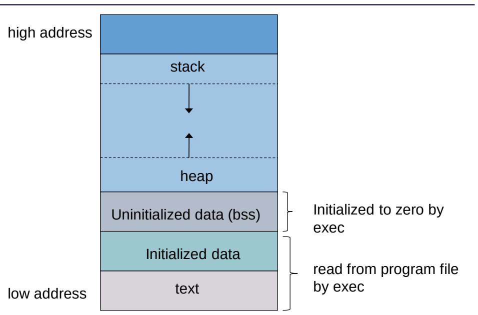
Process Identifiers

```c
/* processid.c get a process information */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
      /* Process ID of calling process */
      printf ("Process ID = %d \n", getpid());
      /* Parent's ID of calling process */
      printf ("Parent's ProcessID = %d \n", getppid());
      /* Real user's ID of calling process */
      printf ("Real User's ID = %d \n", getuid());
      /* Effective user's ID of calling  … */
      printf ("Effective User's ID = %d \n", geteuid());
      /* Real group ID */
      printf ("Real Group's ID = %d \n", getgid());
      /* Effective group ID of calling …*/
      printf ("Effective Group ID = %d \n", getegid());
      return 0;
}
```


The fork() System Call

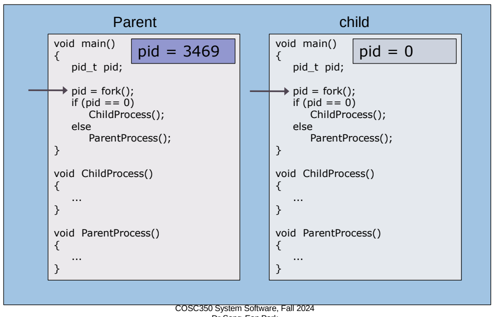
- An existing process can **create a new process** by calling the **fork() system call**.
- The fork() **calls once but returns twice**: a child returns 0 to its parent and a parent returns child's process ID number to the child.
- The child process get a copy of the data space heap and stack and they **don't share the memory space**.
- They only **share the text segment**.

The fork() System Call

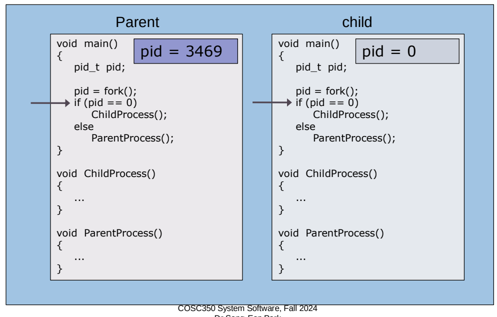



```c
/* fork.c demonstrate fork() system call */
#include  <stdio.h>
#include  <sys/types.h>
#include  <stdlib.h>
#include  <unistd.h>
#define   MAX_COUNT  1000

void  ChildProcess(); /* child process prototype  */
void  ParentProcess(); /* parent process prototype */

void  main(void)
{
     pid_t  pid;
     ppid = getpid(); /* get parent process ID */
     pid = fork(); /* create a child */
     if (pid == 0) /* means a child process*/
          ChildProcess();
     else
          ParentProcess();
}
void  ChildProcess()
{
     int   i;
     for (i = 1; i <= MAX_COUNT; i++)
           printf("   This line is from child process value = %d\n", i);
     printf("   *** Child process is done ***\n");
}
void  ParentProcess()
{
     int   i;

     for (i = 1; i <= MAX_COUNT; i++)
          printf("This line is from parent process  value = %d\n", i);
     printf("*** Parent is done ***\n");
}
```

The fork() System Call



The fork() System Call

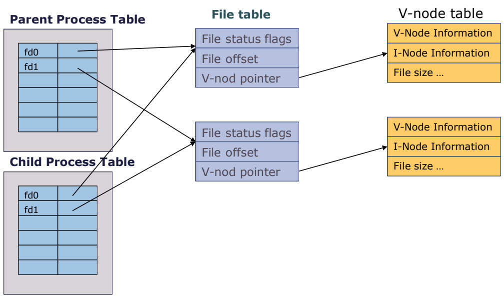



```c
/* fork2.c */
#include  <stdio.h>
#include  <string.h>
#include  <sys/types.h>

#define   MAX_COUNT  200
#define   BUF_SIZE   100

int  main(void)
{
     pid_t  pid, ppid;
     int    i;
     char   buf[BUF_SIZE];
     ppid = getpid(); /* this is parent process ID */
     fork(); /*create a child */
     for (i = 1; i <= MAX_COUNT; i++)
     {
           pid = getpid();
           if (pid == ppid)/* parent works here */
          {
             sprintf(buf, "Parent(%d) process executed %d times\n", ppid, i);
             write(1, buf, strlen(buf));
          }
          else /* child work here */
          {
             sprintf(buf, "Child(%d) process executed %d times\n", pid, i);
             write(1, buf, strlen(buf));
          }
     }
     return 0;

}
```

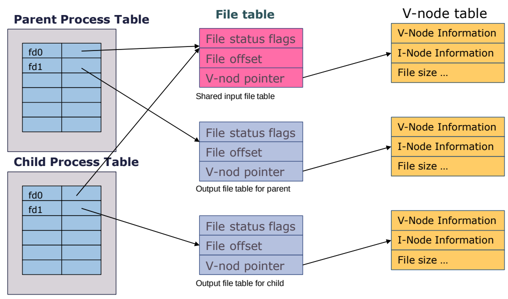

File Sharing

- Consider a process that has two different files opened for input and output.
- On return from fork, parent and child process share file table, since a child copy all from its parent.
- Even offset of file will be shared with both processes.

File Sharing



File Sharing



File Sharing

- If both parent and child write to the same file descriptor, without any form of synchronization what will be happen?
- The output will be intermixed between child and parent's works.
- Solution:
  - The parent waits for the child to complete
  - Both the parent and the child go their own ways


```c
//fileshare1.c
include <unistd.h>
#include <fcntl.h>
#include <ctype.h>
#include  <stdio.h>
#include  <sys/types.h>
#include <sys/stat.h>
void error_sys(char *msg)
{
  printf("%s\n", msg);
  exit(1);
}
int main(int argc, char *argv[])
{

    int input, chout, nbyte, pout;
    int i;
    char buff[1];
    if (argc != 2)
            error_sys("usage: Argument number error\n");

    input = open(argv[1], O_RDONLY);
    pid_t  pid;
     pid = fork(); /* create a child */
     if (pid == 0) /* child process */
     {
         if ((chout = open("child.txt", O_WRONLY|O_CREAT, S_IREAD|S_IWRITE)) == -1)
                 error_sys("Output File Create Error");
         while ((nbyte = read(input, buff, 1)) > 0)
         {
              if (write(chout, buff, 1) != 1)
                   error_sys("Write Error");
         }
      }
     else /* parent */
     {
         if ((pout = open("parent.txt", O_WRONLY|O_CREAT, S_IREAD|S_IWRITE)) == -1)
               error_sys("Output File Create Error");
         while ((nbyte = read(input, buff, 1)) > 0)
         {
               if (write(pout, buff, 1) != 1)
                   error_sys("Write Error");
         }
       }

     return 0;
}
```

File Sharing

- There are two uses for fork:
  - When a process want to duplicate itself so that parent and child can each execute different section of code at the same time – Network Server
  - When a process wants to execute a different program –the child does an exec right after it returns from the forks

The vfork() System Call

- The semantics of vfork() differs from the system call fork().
  - vfork() system call is used to create a new process when the purpose of the new process is to **exec a new program**.
  - vfork() function <u>create a process without copying the address space of the parent</u>.
  - <u>A child runs in the address space of the parent.</u>
  - vfork <u>guarantees that the child runs first</u>, until the child calls exec or _exit – it might lead to deadlock.


```c
/* fork3.c */
#include  <stdio.h>
#include  <stdlib.h>

int glob =6; /*global variable */

int  main()
{
     int local;
     pid_t pid;
     local =88; /*local variable */
     printf ("before vfork\n");
     if ((pid = vfork())<0) /* create a child */
     {
        printf("vfork error");
        exit (1);
     }

     else if (pid == 0) /* for child process */
     {
        glob++;
        local++;
        printf("Child pid = %d, global = %d, local = %d\n",getpid(),glob, local);
        _exit(0);
     }
 /* for parent process */
     printf(" Parent pid = %d, global = %d, local = %d\n",getpid(),glob, local);
     exit(0);
}
```



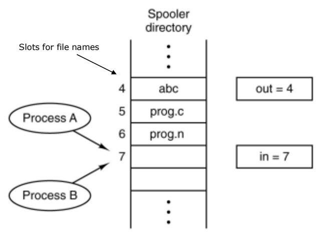
```c
/* fileshare3.c child process run different program
#include <unistd.h>
#include <fcntl.h>
#include  <stdio.h>
#include  <stdlib.h>
#include <sys/stat.h>
void error_sys(char *msg)
{
  printf("%s\n", msg);
  exit(1);
}

int main(int argc, char *argv[])
{
  int input, chout, nbyte, pout;
  int i;
  char buff[1];
   if (argc != 2)
                error_sys("usage: Argument number error\n");

    input = open(argv[1], O_RDONLY);
    pid_t  pid;

     pid = vfork(); /* create a child */
     if (pid == 0) /* child process */
     {
         execv("reverse", argv); /* child process run different program
         _exit(0);
     }
      else /* parent */
      {
         if ((pout = open("parent.txt", O_WRONLY|O_CREAT, S_IREAD|S_IWRITE)) == -1)
                           error_sys("Output File Create Error");
               lseek(input, 0, SEEK_SET);
         while ((nbyte = read(input, buff, 1)) > 0)
                {
                           if (write(pout, buff, 1) != 1)
                                  error_sys("Write Error");
         }
      }

     return 0;
}
```

Race Condition

Race Condition

- A situation where two or more processes are reading or writing some shared data and the final result depends on who runs precisely when, are called race condition.

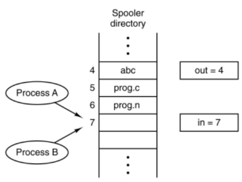
Race Condition



- When a process want to print a file, it enter a file name in a special spooler directory
- Printer daemon periodically check spooler directory any file need to be printed.

Race Condition

- Process A tried to send a job to spooler, Process A read *in = 7*, process A time out and go to ready state before updating *in = in + 1.*
- Process B tried to send a job to spooler. Process B read *in = 7*, load its job name in slot 7, update *in = in + 1 = 8* and then go to block state for waiting for job.
- Process A is rescheduled by scheduler. Process A already read *in = 7*, Process A load its job name in slot 7, update *i = i + 1 = 9* and then go to blocked state waiting for this job finish.




```c
/*race.c : shows example of race condition*/
#include <stdio.h>
#include <stdlib.h>
#include  <string.h>
#include  <sys/types.h>

static void charatatime(char *);

int main()
{
    pid_t pid;
          /* create a child */
    if ( (pid = fork()) < 0)
    {
        printf("fork error");
        exit (1);
    }

    /* a child and parent call same function */
    if (pid == 0)
         charatatime("output from child\n");
    else
         charatatime("output from parent\n");
    exit(0);
}

static void charatatime(char *str)
{
     char *ptr; /* child and parent has its own buffer but using same stdout */
     int c;
     setbuf(stdout, NULL); /*set unbuffered */
     for (ptr =str; c = *ptr++;)
        putc(c, stdout);
}
```