Lecture 3 - Process

Instructor : Bibhas Ghoshal (bibhas.ghoshal@iiita.ac.in)

Autumn Semester, 2015

Lecture Outline

- Process Concept
- Process State
- Process Control Block
- Process Operation
- Inter Process Communication :
  - Shared Memory
  - Message Passing
- Producer/Consumer Problem

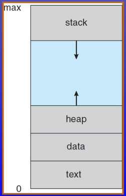
References and Illustrations have been used from:

- lecture slides of the book - Operating System Concepts by Silberschatz, Galvin and Gagne, 2005
- Modern Operating System by Andrew S. Tanenbaum
- lecture slides of CSE 30341: Operating Systems (Instructor : Surendar Chandra),

Process Concept

Process : Program in execution; process execution must progress in sequential fashion; a process is more than a code.
A process includes the following:

- The *text* section (code), *data* section (global variables)
- *Program counter* and contents of registers
- *Stack* to contain function parameters and return addresses (during recursive calls)
- *Heap* required during dynamic memory allocations



Process Concept contd.

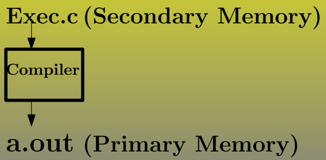


Figure : Program Execution

- Exec.c will reside in secondary storage. OS will pick it up and put it in main memory and execute.
- When OS puts the program in main memory, a process is created.
- The OS maintains a data structure for each process called a Process Control Block

Process Structure

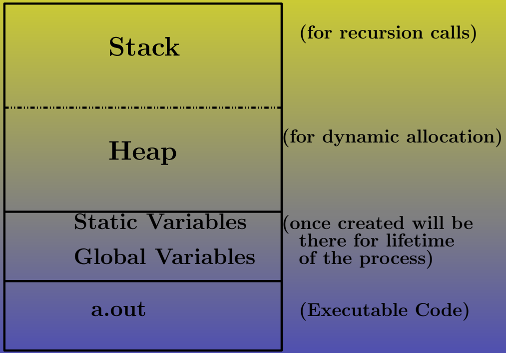


Figure : Process created during execution of a.out

While executing the program, one is restricted within the process boundary, else **Segmentation Fault**

Process Attributes

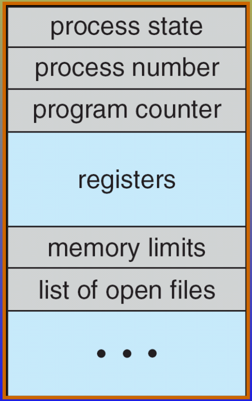
- Process ID : unique number assigned to each process
- Program Counter
- Process State
- Priority
- General Purpose Registers
- List of open files
- List of open devices
- Protection

Process Control Block

Information associated with each process and maintained by the operating system

- Process State
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information

Process Control Block

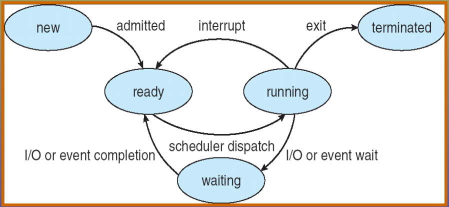


Process State

As a process executes, it changes state

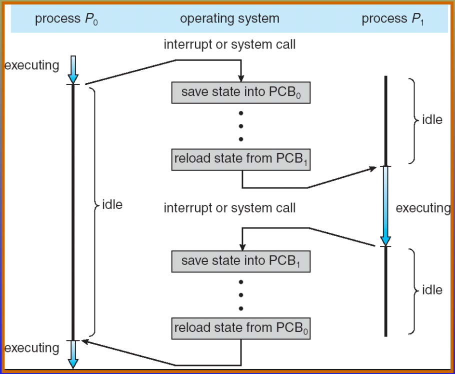
- new: The process is being created (process is in secondary memory)
- ready: process is waiting to be assigned to a processor (process in main memory)
- running: Instructions are being executed (process in main memory)
- waiting: The process is waiting for some event to occur (process in main memory)
- terminated: The process has finished execution (the PCB and all traces of process is deleted)
- suspend ready : processes which were in ready state, but due to lack of resources are backed up in secondary memory
- suspend block : suspend processes which are in blocked/waiting stste and send them to secondary memory

Process State Diagram

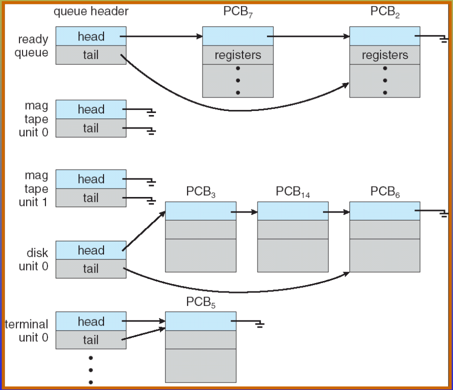


CPU switch from P0 to P1 : Context switching

Save all state of P0, restore all state of P1, save



Ready Queue and other Device Request



Scheduler

- Long-term scheduler (or job scheduler) – selects which processes should be brought into the ready queue
  - invoked very infrequently (seconds, minutes) - (may be slow)
  - should have mix of CPU bound process and I/O bound process
- Short-term scheduler (or CPU scheduler) – selects which process should be executed next and allocates CPU
  - invoked very frequently (milliseconds) - (must be fast)
  - apart from decision making, everything else is done by *dispatcher*
  - dispatcher schedules process with minimum context length
- Medium-term scheduler moves some processes to disk and vice-versa - *swapping*
- Processes can be described as either:
  - I/O-bound process – spends more time doing I/O than computations, many short CPU bursts
  - CPU-bound process – spends more time doing computations; few very long CPU bursts

Numerical example

Consider a system with **N** processors and **M** processes, then the number of processes that will be there in each of the following states are:

State	minimum	maximum
Ready	0	M
Running	0	N
Block	0	M

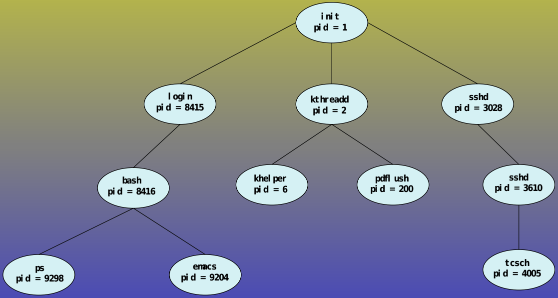
Operations on Process

- Process creation
  - Parent creates new process forming a tree
  - Child process can run concurrently with parent or not
  - Child can share all resources, some or none at all
- Process Scheduling
- Process Execution
- Process termination
  - Exit for normal termination
    - Output data from child to parent (via wait)
    - exit() and _exit() functions
  - Abort for abnormal kernel initiated termination
  - Some OS require the presence of parent to allow child

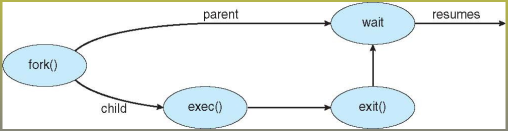
Process Creation

- Parent process create children processes, which, in turn create other processes, forming a tree of processes
- Generally, process identified and managed via a process identifier (pid)
- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate

A Tree of Processes in Linux



Process Creation



C program of fork

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
      fprintf(stderr, "Fork Failed");
      return 1;
    }
    else if (pid == 0) { /* child process */
      execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
      /* parent will wait for the child to complete */
      wait(NULL);
      printf("Child Complete");
    }

    return 0;
}
```

Process Termination

- Process executes last statement and then asks the operating system to delete it using the exit() system call.
  - Returns status data from child to parent (via wait())
  - Process' resources are deallocated by operating system
- Parent may terminate the execution of children processes using the abort() system call. Some reasons for doing so:
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates

Process Termination

- Some operating systems do not allow child to exists if its parent has terminated. If a process terminates, then all its children must also be terminated.
  - cascading termination. All children, grandchildren, etc. are terminated.
  - The termination is initiated by the operating system.
- The parent process may wait for termination of a child process by using the wait()system call. The call returns status information and the pid of the terminated process
  pid = wait(&status);
- If parent terminated (died) without invoking wait, process is an **orphan**
- If no parent waiting (did not invoke wait()) process is a **zombie**
- Zombie process - A child process has completed execution (died) (via the *exit* system call) but still has an entry in the process table. This entry is still needed to allow the parent process to read its child's exit status (using the *wait()* sys call). Then the descriptor is

Example of Zombie process

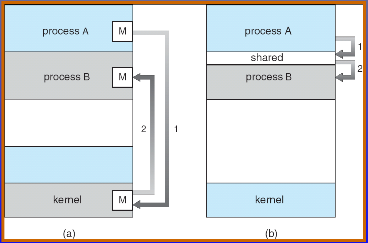
```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main()
{
   pid_t pid, ppid;
   printf("Hello World1\n");
   pid=fork();
   if(pid==0)
   {
      exit(0);
   }
   else
   {
      while(1)
      {
      printf("I am the parent\n");
      printf("The PID of parent is %d\n",getpid());
      printf("The PID of parent of parent is %d\n",getppid());
      sleep(2);
      }
   }
}
```

Interprocess communications

- Independent process cannot affect or be affected by the execution of another process
- Cooperating process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

IPC mechanisms

- Shared memory
  - Create shared memory region
  - When one process writes into this region, the other process can see it and vice versa
- Message passing
  - Explicitly send() and receive()



Producer-Consumer Problem

- Paradigm for cooperating processes, producer process produces information that is consumed by a consumer process
  - **unbounded-buffer** places no practical limit on the size of the buffer
  - **bounded-buffer** assumes that there is a fixed buffer size

Bounded-Buffer – Shared-Memory Solution

- Shared data :

```
#define BUFFER_SIZE 10
typedef struct {
  . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

Bounded Buffer : Producer

```
item next_produced;
while (true) {
    /* produce an item in next produced */
    while (((in + 1) % BUFFER_SIZE) == out)
      ; /* do nothing */
    bu  er[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```

Bounded Buffer : Consumer

```
item next_consumed;
while (true) {
      while (in == out)
            ; /* do nothing */
     next_consumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;
     /* consume the item in next consumed */
}
```

Interprocess Communication – Shared Memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.

Interprocess Communication – Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - **send**(message)
  - **receive**(message)
- The message size is either fixed or variable

Interprocess Communication – Message Passing

- If processes P and Q wish to communicate, they need to:
  - Establish a communication link between them
  - Exchange messages via send/receive
  - Implementation issues:
    - How are links established?
    - Can a link be associated with more than two processes?
    - How many links can there be between every pair of communicating processes?
    - What is the capacity of a link?
    - Is the size of a message that the link can accommodate fixed or variable?
    - Is a link unidirectional or bi-directional?

Wrapup

- Processes are programs in execution
  - Kernel keeps track of them using process control blocks
  - PCBs are saved and restored at context switch
- Schedulers choose the ready process to run
- Processes create other processes
- On exit, status returned to parent
- Processes communicate with each other using shared memory or message passing