Processes and Process Management

Processes

- multiple jobs may be active at the same time
- each job may be a different running program

⇒PROCESS

What is a Process?

Definition:

A process is a sequence of actions resulting from a run of a sequential program written for a specific function.

- process ⇔ task

What is a Process?

- process = a running program
- a process consists of a
  - sequential program code,
  - program counter,
  - register contents,
  - and variables.

What is a Process?

- more than one process for one program
- through system calls, processes
  - use system resources
  - communicate with each other
  - communicate with the world

Program ⇔ Process

Example: A programmer bakes a cake using a recipe.

recipe → program
ingredients → inputs
programmer → processor

Process → programmer reads recipe, obtains ingredients, performs necessary operations.

Program ⇔ Process

(example cntd.) His son enters the kitchen shouting that a bee has stung him. Programmer marks where he left off on the recipe, stops what he is doing, picks up the first aid book and starts the necessary treatment on his son.

- treatment method → program
- medicines → inputs
- programmer → processor

Process → applying first aid using the treatment given in the book

Program ⇔ Process

(example cntd.)

Result: processor shared by two processes in time (time-sharing)

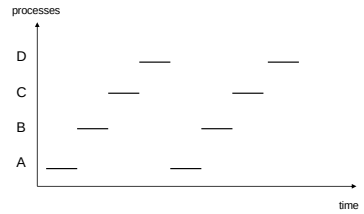
- the process which will have the processor is determined through an algorithm

Processes

- only one processor in system
- time-sharing operation
  - quantum
- only one of each system register
  - program counter, stack pointer, condition code register, general purpose registers, index register, ...

⇒ How is time-sharing achieved?

Time-Sharing

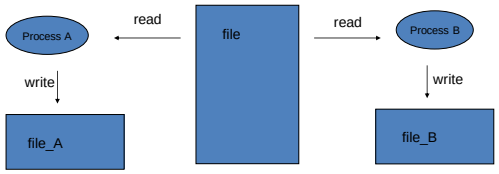


Time-Sharing

- cannot predict when a process will have the processor
  - no time dependent operations in program code!

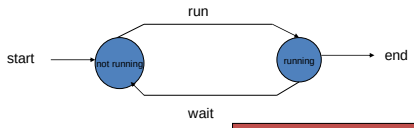
Time-Sharing

Example: what can be said about the contents of *file_A* and *file_B*?



Time-Sharing

- process
  - RUNNING ⇒ has processor
  - NOT RUNNING⇒ does not have processor

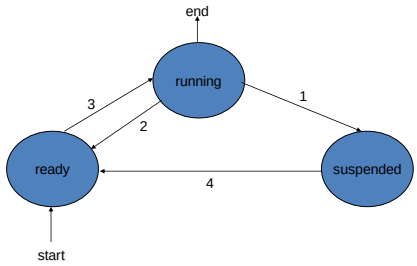


Question: Why would a process wait?

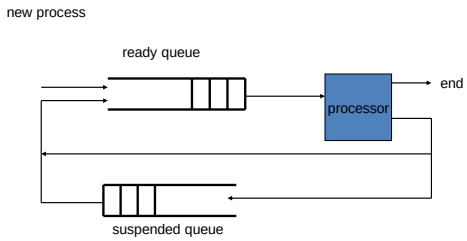
Process States

- processes are in different states throughout their lifetimes
- basic three states
  - running: is using processor
  - ready: can use processor when it gets it
  - suspended: waits for an event; cannot use processor even if it gets it

Process States

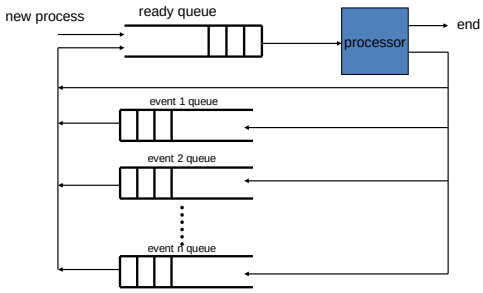


Process States



Question: is ONE queue sufficient?

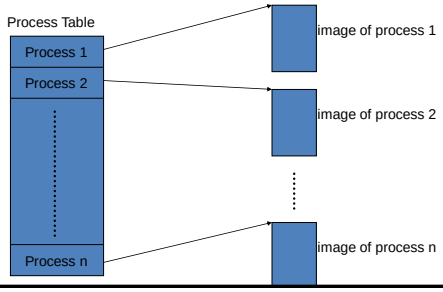
Process States



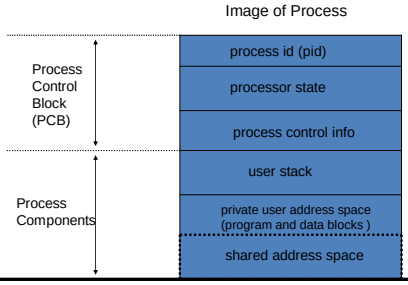
Implementing Processes

- operating system keeps info on all entities it manages
  - a different table for each
    - I/O tables
    - memory tables
    - file tables
    - **proces tables**

Implementing Processes



Implementing Processes



Implementing Processes

- info regarding process in process descriptor field
  - process control block – PCB
    - data that holds info on process
- all operations on process through PCB
  - must have fast access to PCB
    - in some systems through a hardware register
    - in some systems special instructions to access PCB

Process Control Block (PCB)

1. process identification info
   - process id
   - id of process' parent process
   - owner of process
2. current state of process and event it waits for (if any)
3. priority of process
4. scheduling info

Process Control Block (PCB)

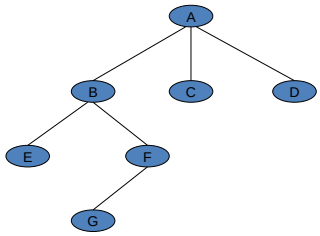
5. pointers to resources used by process
   - e.g. open files
6. pointer to virtual memory allocated to process
7. area where contents of system registers and user accessible (through machine code instructions) processor registers are stored
   - general purpose registers, program counter, condition code register, index register, stack pointer, ...

   ⇒ processor context

Operations on Processes

- create process
  - in UNIX type systems only another process creates a process
  - a hierarchy among processes
  - creator process: parent proses
  - created process: child proses
  - a process may create multiple child processes

Operations on Processes



Operations on Processes

- when a process is created:
  - if process table full, process NOT created
  - if process table entry available,
    - process is assigned a unique id
    - process is assigned initial priorities
    - PCB is created and initialized
    - initial resources assigned (memory etc)
    - process is added to *ready* queue

Operations on Processes

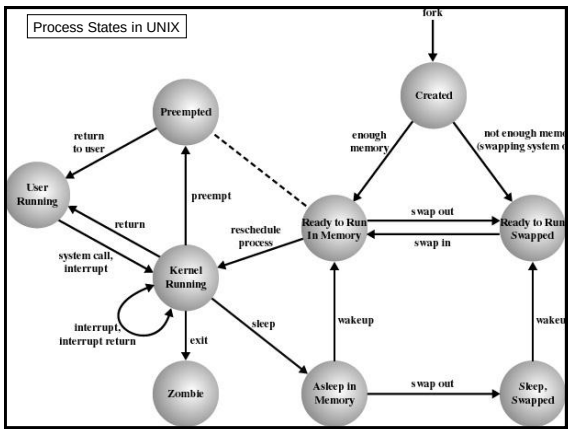
- destroy a process
  - process removed from system
  - resources returned to system
  - pid returned to system
  - PCB and process table entry deleted
  - necesssary operations on children performed
    - either keep entry until all children exit
    - or children assigned to another parent
      - e.g. in UNIX to the *init* process (pid=1)

Operations on Processes

- suspend a process
  - for short term suspension, resources not removed
  - for long term suspension, depending on resources, some may be removed
- resume a process
  - to resume operation of a process from where it left off in its previous execution
- change priority of a process

Steps Performed during a Process State Change

- processor context saved
- PCB of running process updated
- running process added to appropriate queue (ready / suspended)
- new process to run determined
- PCB of selected process updated
- info on memory management updated
- context of selected process loaded onto registers



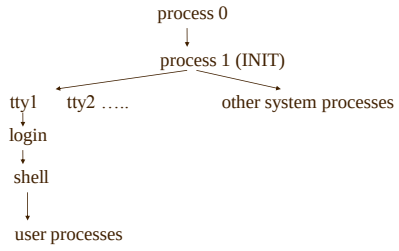
Process Creation in UNIX

- fork system call
  - parent process
  - child process
- syntax: *pid=fork()*
  - both processes have same context
  - returns pid of child to parent process
  - returns 0 to child process
- process no. 0 is the only process not created using *fork*

Process Creation in UNIX

- when fork system call is made:
  - if possible, reserve entry in process table (max no of processes)
  - assign unique pid to child process
  - make a copy of the context of the parent process
  - file access counters modified
  - return child pid to parent process and 0 to child process

Process Hierarchy in UNIX



Exit System Call

- ends execution of process
- syntax: *exit(status)*
  - "*status*" returned to parent process
- all resources returned
- file access counters modified
- process table entry deleted
- when a parent process exits all its children are assigned the init process as a parent (process no 1)

Sample program code - 1

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

int f;

int main (void)
{
  printf("\n Program running: PID=%d \n",
           getpid());
  f=fork();
```

Sample program code - 2

```
if (f==0)  /*child*/
  {
    printf("\nChild process: my pid = %d\n",
            getpid());
    printf("Child process: my parent pid = %d\n",
            getppid());
    sleep(2);
    exit(0):
  }
```

Sample program code - 3

```
else /* parent */
  {
    printf("\nParent process: my pid = %d\n",
            getpid());
    printf("Parent process: my parent pid = %d\n",
            getppid());
    printf("Pranet process: my child's pid = %d\n",
            f);
    sleep(2);
    exit(0):
  }
  return(0);
}
```