Processes

Bojan Nokovic

Based on: "Operating Systems Concepts", 10th Edition Silberschatz Et al.
"Slides 3SH3 '12" - Sanzheng Qiao

Jan. 2021

Process Concept

Early computers were batch systems that executed jobs.

User programs run by time-shared systems are called tasks.

Even if computer executes only one program, OS may need to support its own internal activities, such as memory management.

What is a Process?

Program in execution is the most frequently referenced one. Process execution must progress in sequential fashion.

Is a process the same as a program?

- Program becomes process when executable file loaded into memory
- One program can be several processes (i.e. multiple users executing the same program)
- Program is passive entity stored on disk (executable file); process is active

Address space

Each process is associated with an address space.

All the state needed to run a program (execution stack, system environment, etc.). It contains all the addresses that can be touched by the program.

Why address space? Protection. A process can only access its own address space.

A process can itself be an execution environment for other code

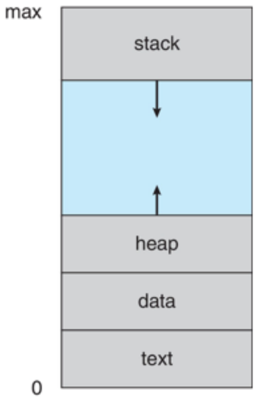
```
java Program
```

A process is represented by its Process Control Block (PCB):

- Address space.
- Execution state (PC, saved registers).

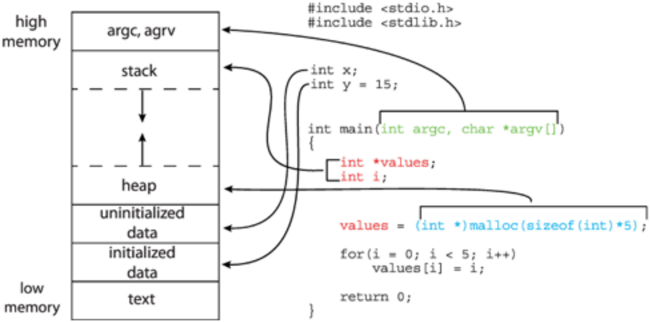
Process in Memory

- Text - the executable code
- Data - global variables
- Heap - dynamically allocated memory
- Stack - temporary data storage when invoking functions (such as function parameters, return addresses, and local variables)



The stack and heap sections grow toward one another, how to ensure they do not overlap one another?

Memory Layout of a C Program



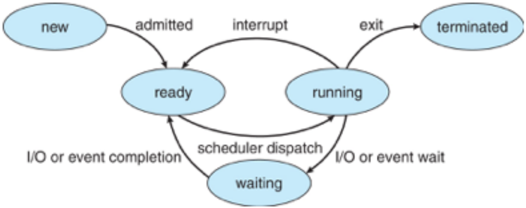
Process States

As a process executes, it changes state

- New: Just created
- Waiting: Waiting for an event to occur.
- Ready: Has acquired all the resources but the CPU.
- Running: Running on the CPU.
- Finish: Exiting.

Processes switch from one state to another, OS controls this.

Diagram of Process States



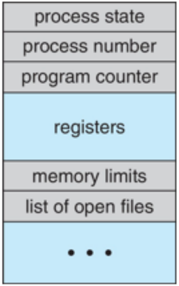
Only one process can be running on any processor core at any instant, however many processes may be ready and waiting.

Deterministic or nondeterministic process?

Process Control Block (PCB)

Each process is represented in the operating system by a PCB

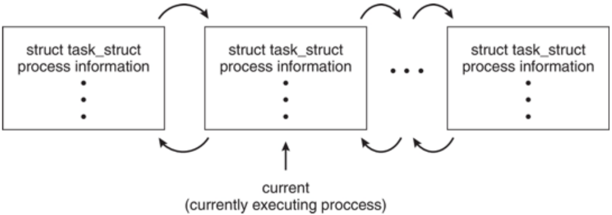
- Scheduling information (priority).
- Accounting information (CPU time).
- Open files.
- Other miscellaneous information.



OS maintains a process table (a collection of all PCBs) to keep track of all the processes.

Process Representation in Linux

```
pid t_pid;                          /* process identifier */
long state;                         /* state of the process */
unsigned int time_slice             /* scheduling information */
struct task_struct *parent;         /* this process's parent */
struct list_head children;          /* this process's children */
struct files_struct *files;         /* list of open files */
struct mm_struct *mm;               /* address space of process */
```



If `current` is a pointer to the process currently executing, its state is changed with the following:

```
current->state = new_state;
```

Process Scheduling

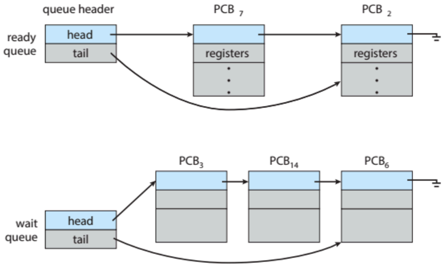
Maximize CPU use, quickly switch processes onto CPU core

Process scheduler selects among available processes for next execution on CPU core

Maintains scheduling queues of processes

- Ready queue - set of all processes residing in main memory, ready and waiting to execute
- Wait queues - set of processes waiting for an event (i.e. I/O)
- Processes migrate among the various queues

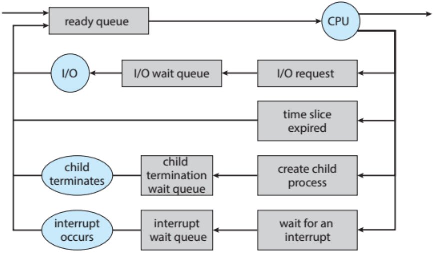
Ready and Wait Queues



As processes enter the system, they are put into a ready queue. It waits there until it is selected for execution, or dispatched.

Processes that are waiting for a certain event to occur - such as completion of I/O - are placed in a wait queue.

Representation of Process Scheduling



The process could issue an I/O request and then be placed in an I/O wait queue.

The process could create a new child process and then be placed in a wait queue while it awaits the child's termination.

The process could be removed forcibly from the core, as a result of an interrupt or having its time slice expire, and be put back in the ready queue.

Dispatcher

With many processes on the the system, OS must take care of:

- Scheduling: each process gets a fair share of the CPU time.
- Protection: processes don't modify each other

Dispatcher:

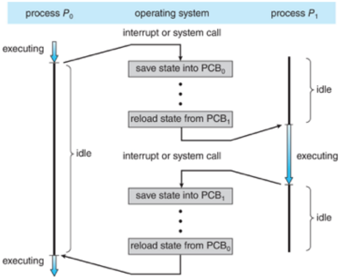
1. Run process for a while
2. Pick a process from the ready queue
3. Save state (PC, registers, etc.)
4. Load state of next process
5. Run (load PC register)

Dispatcher

When a user process is switched out of the CPU, its state must be saved in its PCB. Everything could be damaged by the next process:

- Program counter.
- Processor status word.
- Registers (General purpose and floating-point).

CPU Switch From Process to Process



Switching the CPU core to another process requires performing a state save of the current process and a state restore of a different process.

This task is known as context switch.

The kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run.

Exceptions

The CPU can run only one process at a time. When a user process is running, the dispatcher (part of OS) is not running.

How can OS regain control of the CPU?

- Exceptions: User process gives up the CPU to OS (caused by internal events, for example, go to sleep)
  - System call.
  - Error (eg. bus error, segmentation error, overflow, etc.).
  - Page fault.
  - Yield.

These are also called traps.

Interrupts

The OS interrupts user process (caused by external events):

- Completion of an input eg. a character typed at keyboard.
- Completion of an output - a character displayed at terminal.
- Completion of a disk transfer.
- A packet is sent to the network.
- Timer (alarm clock).

Operations on Processes

Process creation

Process termination

Process creation

Creating a process from scratch:

1. Load code and data into memory.
2. Set up a stack.
3. Initialize PCB.
4. Make process known to dispatcher.

Process Creation

Forking a process:

1. Make sure the parent process is not running and has all state saved.
2. Make a copy of code, data, and stack.
3. Make a copy of PCB of the parent process into the child process.
4. Make the child process known to dispatcher.

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)

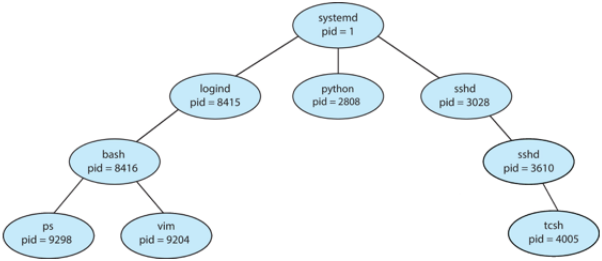
Resource sharing options

- 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



The `systemd` process (which always has a pid of 1) serves as the root parent process for all user processes.

The `systemd` process creates processes which provide additional services.

List of processes: `ps -el`

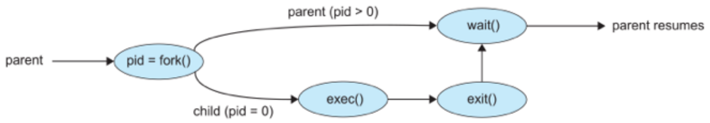
Process Creation

Address space

- Child duplicate of parent
- Child has a program loaded into it

UNIX examples

- `fork()` system call creates new process
- `exec()` system call used after a `fork()` to replace the process' memory space with a new program
- Parent process calls `wait()` for the child to terminate



Example

UNIX `fork()`, `exec()`, and `wait()`

The system call `fork()` is called by one process and returned in two processes.

Parent: returns child pid, Child: returns 0

```
pid = fork();
if (pid == 0) /* child process */
        exec("executable");
/* parent process continues */
```

In the child process, `executable` overwrites the old program.

Parent process calls `wait()` for the child to terminate

C Program Forking Separate Process

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

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

Terminating when it finishes the last statement and calls `exit()`

- Deallocate memory (physical and virtual)
- Close open files
- Notify its parent process (via `wait()`)

Process Termination

Terminated by another process, usually the parent, using system call `abort()` or `kill()`.

- 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.

- cascading termination. All children, grandchildren, etc. are terminated.

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);
```

A process that has terminated, but whose parent has not yet called `wait()`, is known as a zombie process.

If parent terminated without invoking `wait()`, process is an orphan

Interprocess Communication

Processes within a system may be independent or cooperating

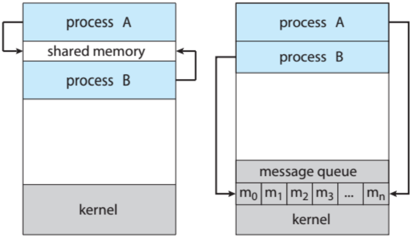
Cooperating process can affect or be affected by other processes, including sharing data

Reasons for cooperating processes:

- Information sharing
- Computation speedup
- Modularity
- Convenience

Cooperating processes need interprocess communication (IPC)

Communications Models



Two models of IPC:

- shared memory
- message passing

Thank You

Operating Systems are among the most complex pieces of software ever developed !