CS415 Module 1 Part A - Introduction

Athens State University

Contents

1 What is a process?

Definition of "process"

- A program in execution
- An instance of a program running on a computer
- The entity that can be assigned to and executed on a processor
- A unit of activity
  - Executing a sequence of instructions
  - Working upon a current state
  - Using a set of system resources

Program vs. Process

- **Program**: A file containing information that describes how to construct a process at run time
- This is the final output of the *compile-link-debug* cycle
- The program file contains information about what format is being used, the machine language instructions, data, shared-library and dynamic-linking information
- One program may be used to construct many processes and many processes may be running the same program

Program files are a binary file format that defines how to construct a process. This information includes:

- **Binary format identification**: Each program file includes meta-information describing the format of the executable file. This enables the kernel to interpret the remaining information in the file.
  - Historically, two widely used formats for UNIX executable files were the original a.out ("assembler output") format and the later, more sophisticated COFF (Common Object File Format).

  - Modern UNIX implementations (including Linux) employ the Executable and Linking Format (ELF), which provides a number of advantages over the older formats.

- **Machine-language instructions**: These encode the algorithm of the program.
- **Program entry-point address**: This identifies the location of the instruction at which execution of the program should commence.
- Data: The program file contains values used to initialize variables and also lit- eral constants used by the program (e.g., strings).
- **Symbol and relocation tables**: These describe the locations and names of functions and variables within the program. These tables are used for a variety of purposes, including debugging and run-time symbol resolution (dynamic linking).
- **Shared-library and dynamic-linking information**: The program file includes fields listing the shared libraries that the program needs to use at run time and the pathname of the dynamic linker that should be used to load these libraries.
- **Other information**: The program file contains various other information that describes how to construct a process.

Process Elements

- Two essential elements
  - Program code: that may be shared with other processes executing the same program
  - A set of data associated with that code

A process is the entity that is executing program code to change the state of the associated data

Process Elements

- *Process*: an execution stream in a the context of a particular process state
- *Execution stream*: a stream of machine instructions
- *Process state*: determines effect of running code
  - Registers: general purpose, program counter, ...
  - Memory: everything a process can address: code, data, stack, heap
  - I/O status: file descriptor table, ...

Process Id and Parent Process Id

- Process ID: a positive integer that uniquely identifies the process on the system
- Parent Process ID: The Process ID of the process that created this process
- Format and structure of process IDs
  - Process IDs range between 1 and 32,767.
  - Each new process is assigned the next sequential process ID
  - When the system reaches 32,767, the process ID rolls over to 300

Why reset to 300? Low-numbered process IDs are in permanent use by system processes and daemons. So, resetting the counter to 300 avoids performance issues with trying to find unused numbers in that range.

It's important to understand that, with the exception of a few important system processes, there is no fixed relationship between a program and the process ID of the process that is created to run that program.

One can find the parent process ID for any process by examining the `Ppid` field in the Linux system file `/proc/PID/status`.

There are two system calls that you use to get the PID and PPID of a process:

```c
#include <unistd.h>
pid_t getpid(void);

pid_t getppid(void);
```

2 Why use processes?

Why use processes?

- Expression of concurrency: System has many concurrent jobs executing
- General principle of divide and conquer: Decompose large problem into smaller ones, easier to think of well contained smaller programs
- Isolated from each other: With well-defined interactions

System Categorization

- Uniprocessing: one process at a time
  - Example: early mainframes, MS-DOS
  - Good: Simple
  - Bad: Poor resource utiliztion
- Multi-processing: multiple processes; when one waits, switch to another
  - Examples: modern operating systems
  - Good: increased resource utilization
  - Bad: complex

Multi-processing

- The OS must support
  - Scheduling: what process to run?
  - Dispatching: how to switch?
  - Memory protection: how to protect one process from another process
- Recall: Separation of policy and mechanism
  - Policy: decision making based on some performance metric and workload $\rightarrow$ scheduling
  - Mechanism: Low-level code to implement decisions about dispatching, protection

3 The Structure of a Process

Process Memory Layout

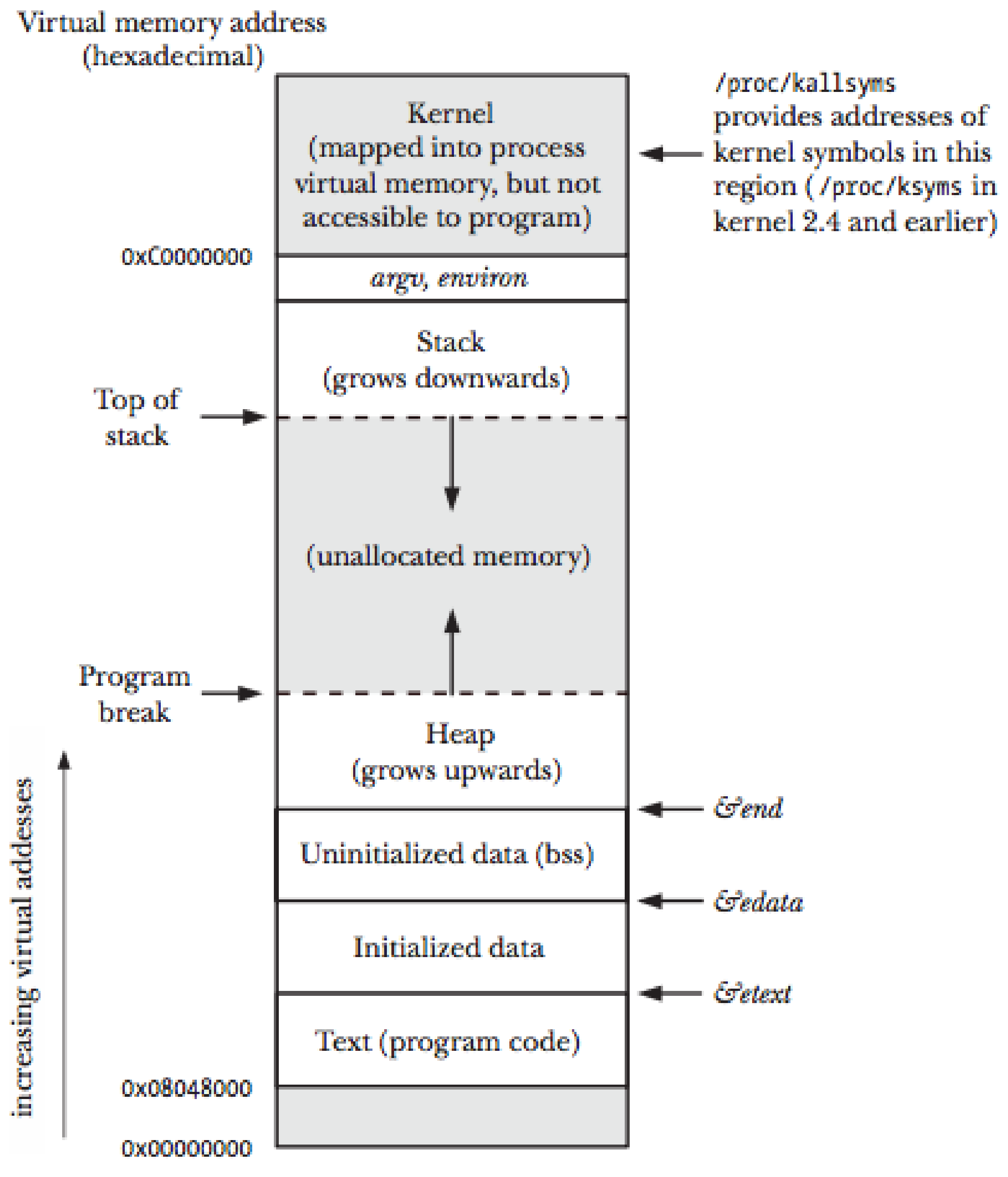

Virtual memory address (hexadecimal)
Kernel (mapped into process virtual memory, but not accessible to program)
/proc/kallsyms provides addresses of kernel symbols in this region (/proc/ksyms in kernel 2.4 and earlier)
0xC0000000
argv, environ
Stack (grows downwards)
Top of stack
(unallocated memory)
Program break
Heap (grows upwards)
&end
Uninitialized data (bss)
&edata
Initialized data
&etext
Text (program code)
increasing virtual addresses
0x08048000
0x00000000

Figure 6-1: Typical memory layout of a process on Linux/x86-32

- This figure shows how a process is allocated in virtual memory
- Note how the heap grows upwards while the stack grows downward
- That space at start of the map contains the *process control block*

The *text* segment contains the machine-language instructions of the program run by the process. The text segment is made read-only so that a process doesn't accidentally modify its own instructions via a bad pointer value. Since many processes may be running the same program, the text segment is made sharable so that a single copy of the program code can be mapped into the virtual address space of all of the processes.

The *uninitialized data segment* contains global and static variables that are not explicitly initialized. Before starting the program, the system initializes all memory in this segment to 0. For historical reasons, this is often called the bss segment, a name derived from an old assembler mnemonic for "block started by symbol." The main reason for placing global and static variables that are initialized into a separate segment from those that are uninitialized is that, when a program is stored on disk, it is not necessary to allocate space for the uninitialzed data. Instead, the executable merely needs to record the location and size required for the uninitialized data segment, and this space is allocated by the program loader at run time.

The *stack* is a dynamically growing and shrinking segment containing stack frames. One stack frame is allocated for each currently called function. A frame stores the function's local variables (so-called automatic variables), arguments, and return value.

The *heap* is an area from which memory (for variables) can be dynamically allocated at run time. The top end of the heap is called the program break.

Locations of variable in process segments

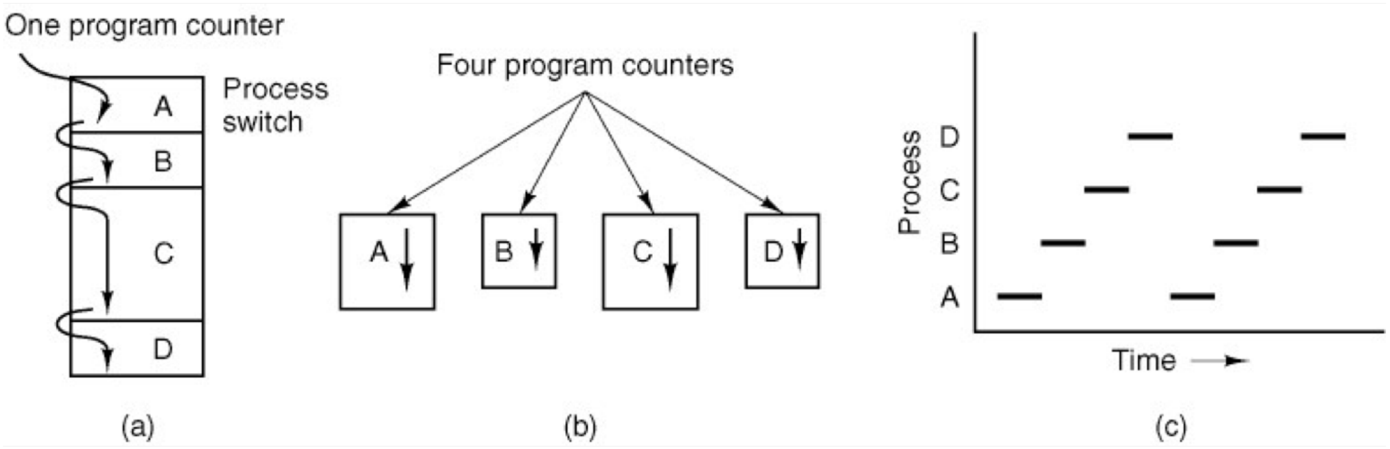
```c
#include <stdio.h>
#include <stdlib.h>
char globBuf[65536];          /* Uninitialized data segment */
int primes[] = { 2, 3, 5, 7 }; /* Initialized data segment */
static int square(int x)
    int result;          /* Allocated in frame for square() */
    result = x * x;
    return result;      /* Return value passed via register */

}
static void doCalc(int val)
{
    printf("The square of %d is %d\n", val, square(val));
    if (val < 1000) {
      int t;           /* allocated in frame for docCalc() */
      t = val * val * val;
      printf("The cube of %d is %d\n", val, t); }
}

int main(int argc, char *argv[])
{
    static int key = 9973;      /* Initialized data segment */
    static char mbuf[10240000];   /* Uninitialized data seg.*/
    char *p;                    /* Allocated in frame for main()*/
    p = malloc(1024);           /* Points to memory in heap      */
    doCalc(key);
    exit(EXIT_SUCCESS);
}
```

Role of the Process Control Block

- The most important data structure in the OS
  - Contains all of the information about a process that OS uses
  - PCB is used by virtually every module in the OS
  - Defines the state of the OS

- Difficulty is not access, but protection
  - A bug in a single routine can damage PCBs and cause the OS to be unable to manage affected processes
  - A design change in the structure or semantics of the PCB could affect a number of modules in the OS

One Program Counter



Process Creation

- **Process spawning**: when the OS creates a process at the explicit request of another process
- **Parent process**: Original process
- **Child process**: New process

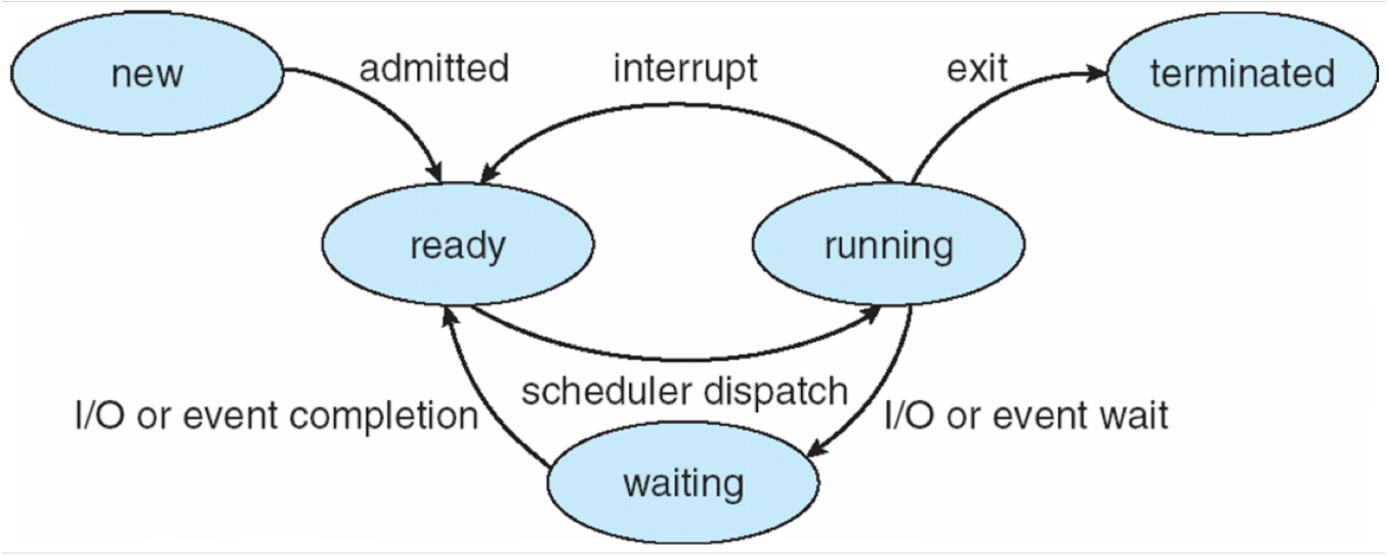
Process Termination

- There must be a means for a process to indicate that it has completed
- Batch jobs will include a `HALT` instruction or system call for termination
- Interactive applications have some user initiated action that indicates process termination

Why do processes terminate

- Four ways for a process to terminate
  - Normal exit (voluntary)
  - Error exit (voluntary)
  - Fatal exit (involuntary)
  - Killed by another process (involuntary)

Process State



Process State Trace

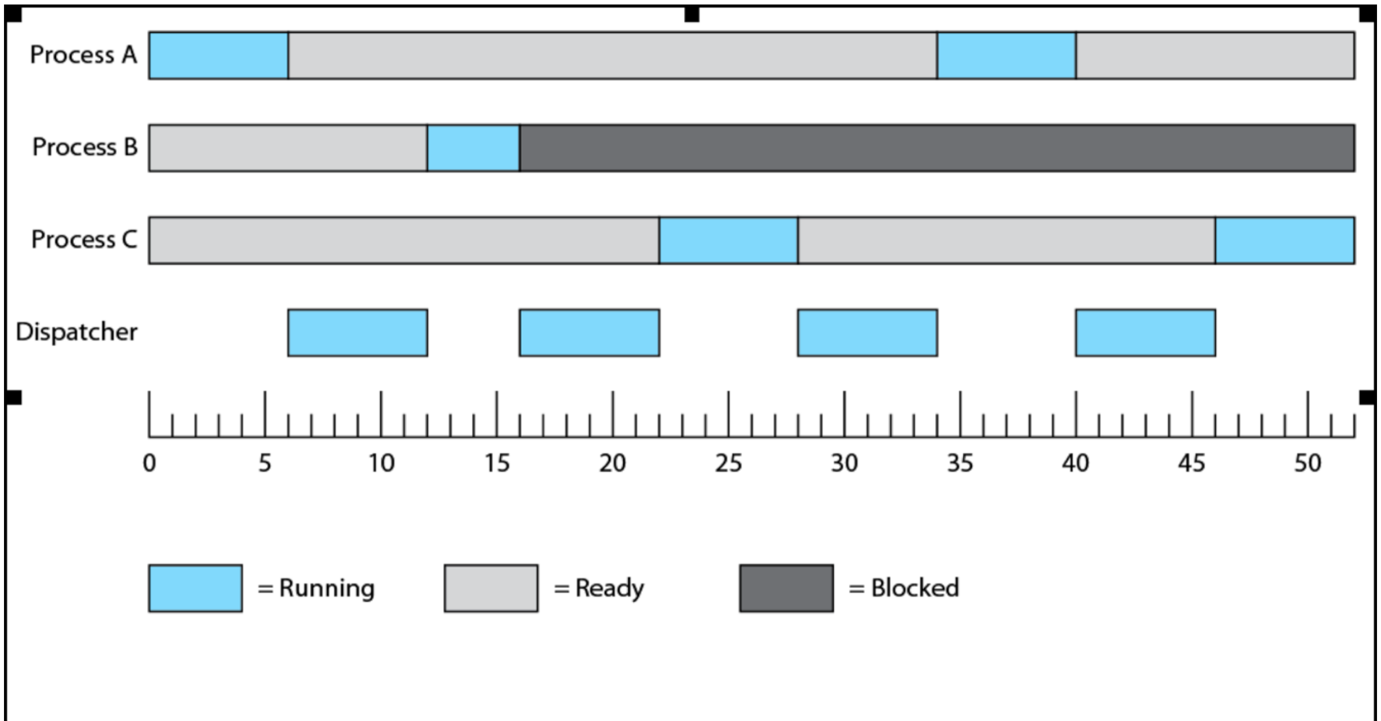


Figure 3.7 Process States for Trace of Figure 3.4

Suspended Process

- Swapping
  - involves moving part of all of a process from main memory
  - when none of the processes in main memory is in the Ready state, the OS swaps one of the blocked processes off of the disk into a suspend queue

Characteristics of a Suspended Process

- The process is not immediately available for execution
- The process was suspended by an agent
  - By itself
  - It's parent process
  - Or the operating system
- The process may or may not be waiting on an event
- The process may not be removed from this state until the agent that suspended the process explicitly orders the removal

Reasons to suspend a process

- Swapping: OS needs to release sufficient main memory to bring in a new process
- Other OS reason: OS suspends a process causing problems
- Interactive user request: Use may need to suspend execution of a program
- Timing: Process may be executed on a periodic basis and may suspend while waiting for the next time
- Parent process request: parent process may need to suspend a child process

4 Key Points

Key Points

- The most fundamental concept in a modern OS is the process
- The principal function of the OS is to create, manage, and terminate processes
- Process control block contains all of the information required to manage the process
  - Current state, resources allocated to the process, priority, and other relevant data
- The most import states are Ready, Running, and Blocked
- The running process is the one that is currently being executed by the processor
- A blocked process is waiting for the completion of a some event
- A running process is interrupted either by an interrupt or by executing a privileged call to the OS