Chapter 3: Processes

3.1 Process Concept

3.1.1 The Process

Process: a program in execution, can be part of the operating system or a user program, but it's not a program. A program becomes a process when it's loaded into memory.
Composed of:

- Text section: program code.
- Current activity: copy of the program counter and registers.
- Stack: temporary data.
- Data section: global variables.
- Heap: dynamically allocated memory.

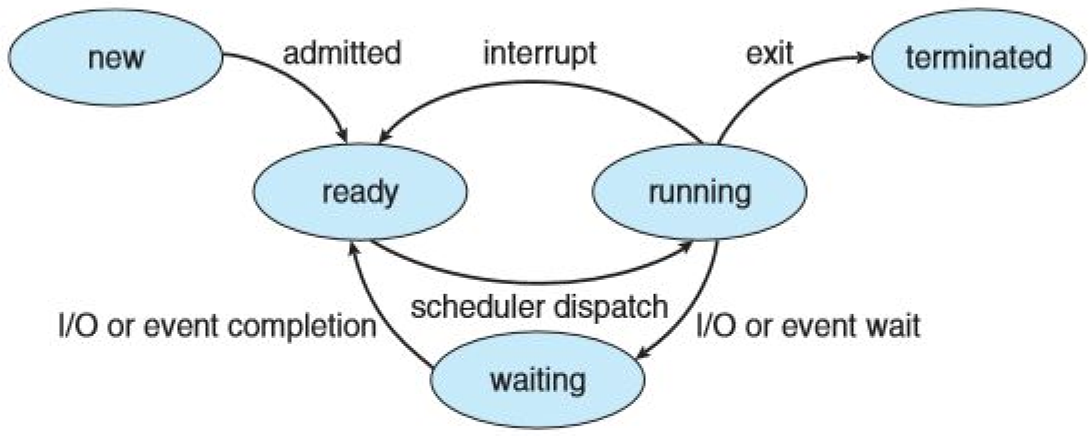
Processes may come from the same program, but they are different.

3.1.2 Process State

The state of a process changes during execution, possible states are:

- New: process is being created.
- Running: instructions are running.
- Waiting: process is waiting for an event to happen (like I/O operations).
- Ready: process is waiting to be assigned to a processor.
- Terminated: process has finished execution.

Note: only one process can run on a processor. These states run in the following loop:



3.1.3 Process Control Block

Also known as task control block, it represents a process in the O.S. and contains more information about the process, such as:

- Process state.
- Program counter: indicates the address of the next instruction to execute.
- CPU Registers: a copy of all the registers at the time of the last interruption.
- CPU-scheduling information: process priority, pointers to scheduling queues and other parameters.

- Memory-management information: info like value of base and limit registers and page tables, or segment tables.
- Accounting information: keeps tabs on information like amount of CPU/real time used, time limits, account numbers, job/process numbers, etc.
- I/O status information: I/O devices allocated to the process, list of open files, etc.

3.1.4. Threads
Program runs on a single thread of execution. What if we make it so that the O.S. handles multiples threads, it's useful for multi-processor units, and running multiple threads in parallel.

3.2. Process Scheduling
Time sharing becomes a thing: how do you give all processes a bit of processor time to make it look like multitasking? Process scheduler takes over and takes an available process out of the many and puts it on a processor.

3.2.1. Scheduling Queues
There are various queues used by the operating system to keep everyone happy:
- Job queue, all processes are put here.
- Ready queue: where "ready" processes hold until being put into a CPU.
- Device queue: processes waiting for a specific I/O device.

3.2.2. Schedulers
Since processes change constantly in between these queues, schedulers have to choose processes from them in order to keep them flowing. Two types of schedulers exist on batch systems: long-term (job) scheduler, a slower scheduler that loads processes from mass-storage devices onto memory, for execution; and the much faster short-term (CPU) scheduler, which loads processes from memory to the CPU.
The long-term scheduler must be careful when choosing. Two types of process exist, I/O bound (spends most of the time in I/O operations) and CPU bound (spends most of the time in CPU operations), so the long-term scheduler must make sure to make a good mix of these processes: too many I/O processes and the ready queue will be empty and the CPU doing little, while too many CPU processes imply the I/O queue and the devices to be doing nothing.
Some systems introduce the middle-term scheduler, which swaps out some processes from the memory pool to reduce the number of competing processes in memory and CPU.

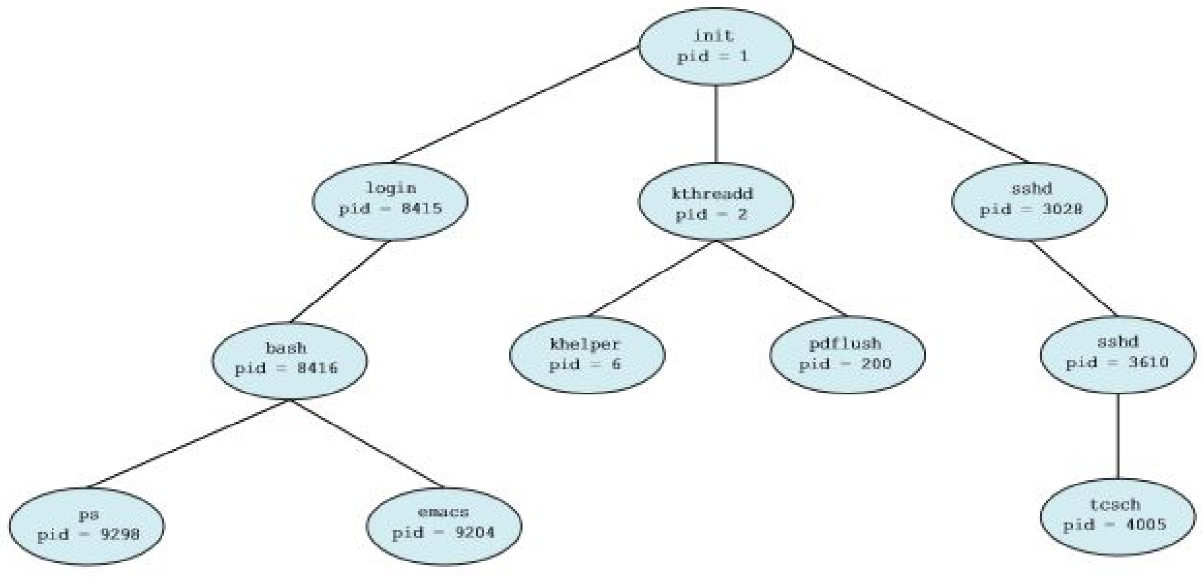
3.2.3.Context Switch
Since processes have to be oftenly swapped out from the processor to run other stuff, the state of the process must be able to be saved (state save) or restored (save restore). Switching the CPU to another process involves saving the state, in a process known as context switch, saving the old process' data and loading the new process' data. This time is purely overhead, nothing productive is done in context switching.

3.3. Operations on Processes

Processes in most systems can run concurrently, and may be created/deleted dynamically. Operating systems must offer mechanisms to create/terminate processes.

3.3.1. Process Creation

A process may spawn child processes, and they in turn can do the same, forming a tree of processes. Most OSs identify a process with an unique process identifier (pid). In *NIX, a tree of processes (or tasks per UNIX naming conventions) may look like this:



When a process spawns a child, the latter must also receive some resources to complete its task, that can be given out by the OS or using a subset of the parent's resources, this last option created with the idea of not overloading the system by creating too many children that would hog the resources. As well as resources, the parent must also pass necessary input into the child process.

When a process spawns children, two things may happen:

- Parent continues execution concurrently with its children.
- Parent waits until some/all children have terminated.

There's also two address-space possibilities for the children:

- It is an exact copy of the parent process (same program and data).
- It loads a new program into it.

To exemplify the difference, let's look at UNIX:

UNIX

UNIX keeps a process id (PID) on all processes. To create a new process, we use the fork() syscall, which creates a copy of the address space of the original process, allowing for easy communication between parent and child. **Both parent and child continue the execution of the code *after* the fork() call, with one difference: the return value of fork() for the child process is zero, but for the parent, it returns the PID of the child**.

After fork(), two things can happen:

- One of the processes calls exec() or execlp(), which loads a new binary program into its memory space and replaces the original code.
- Both parent and child continue the execution of the same code.

3.3.2. Process Termination

Typically, a process ends when it finishes executing its last statement and asks the OS to delete it by using the appropriate system call (like exit() or TerminateProcess()), and all the resources associated with the process are deallocated by the OS.
Alternatives to termination include the possibility of being terminated by another process, such as the parent process, for reasons like:

- Child has exceeded the usage of assigned resources.
- Task assigned to the child is no longer required.
- Parent is exiting, and the OS does not like parentless children.

For the last point, some OSs do not allow children to run without their parents, so *cascading termination* of the children is initiated by it.
In *NIX, using exit(code) terminates a process. A parent process waiting for the termination of a children can listen for the PID and return code like this:

```
pid_t pid;
int status;
pid = wait(&status);
```

When processes end, all but one resource is freed: the entry on the process table. This entry is kept in there with the PID and return code until the parent calls for wait(), and until then, it is a *zombie process*. If the parent process never calls wait(), and instead terminates, the process is said to be an orphaned process. *NIX resolves this by assigning the mother of all processes, init, as parent to orphan processes, who periodically invokes wait() on said processes.

3.4. Interprocess Communication

Processes that do not affect other processes, or share any memory, are said to be *independent*, whereas processes that *do* affect other processes are said to be *cooperating*. Reasons to allow for cooperation are:

- Information sharing: several users are interested in the same piece of info.
- Computation speedup: breaking up a process into parallel subtasks is a good way to speed up a task.
- Modularity: we want our system to be modular, diving system functions into separate processes or threads.
- Convenience: an individual user may be working on multiple tasks at a time.

To communicate, they need an interprocess communication mechanism, it can be either:

- Shared memory: a region of memory is shared by the cooperating processes, where they can read/write data on this region, faster than message passing.
- Message passing: messages are sent and received by cooperating processes. Good for small passage of data, better in distributed systems.

Shared-Memory Systems

- Processes must declare a common memory area, usually on the address space of the creator process, the others just attach.
- Both processes must agree on sharing the memory, otherwise it may end on a segfault.
- Memory management is conceded to the processes, they are responsible for the location and form of data, and to ensure that no two processes are writing to the same memory section.
- Buffers can be created in shared memory:
    - Unbounded buffer: no size limit.
    - Bounded buffer: size limit.
- Synchronization of the memory is important. Check chapter 5 for more info.

Message-Passing Systems

- Create a way to send messages between processes.
- Two operations: send(message) and receive(message).
- Fixed message length and variable message length.
    - Fixed: easy to implement, harder to use (need to fit everything in the size limit).
    - Variable: harder to implement, easier to use (any message can be sent).
- For process to communicate, a connection must exist, and it should define if the communication is:

  Direct or indirect:
    - Directly send a message: The sender must know the receiver, but the receiver may not need to know who sent it. Problematic, as it can cause hard-coding values for checking who sent me what/who do I send what.
    - Indirectly send a message: Introduces the concept of mailbox/port, each one with its unique ID. The mailbox is shared between various processes, where a process can send a message and either:
        - Only one process is able to receive the message.
        - Force the receive() function to only work on one recipient at a time, but all can access it.

  Synchronization, needed to coordinate calls of send() and receive(). Things to consider are:
    - Blocking send: either block the sending process until the message is read, or continue operations.
    - Blocking receive: either block the receiver until a message is read, or constantly read either a valid message or a null.

  Buffering, indifferent to the method of sending, the messages are left on a buffer for sending:
    - Zero capacity: no queue, so the sender must lock until the message is received.
    - Bounded capacity: at most, *n* messages can be stored. In case *n* messages are stored and a new one needs to enter, the sender locks until space is made.
    - Unbounded capacity: possible infinite messages, sender never blocks.

3.5 Examples of IPC Systems

3.5.1 POSIX: Shared Memory

- Process creates the shared memory segment: `shm_fd = shm_open(name, O_CREAT | O_RDRW, 0666);`
- Used to open existing segments for sharing.
- To set the size of the object, in bytes: `ftruncate(shm_fd, SIZE); // SIZE is defined as 4096`
- Process can write to the memory as if it were an output channel:
- Producer:
    - `ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0); // Get a pointer to the start of the memory`
    - `sprintf(ptr, "Writing to shared memory"); // Write to memory`
    - `ptr += strlen(message); // Move the pointer to the next memory address`

3.5.2 Mach: Message based

- Syscalls are messages as well.
- All tasks have two mailboxes by default: kernel and notify.
- Only 3 syscalls needed: `msg_send(), msg_receive(), msg_rpc().`
- Create a new mailbox by using `port_allocate().`
- Send/receive is flexible, four options exist if the mailbox is full:
    - Wait indefinitely.
    - Wait, at most, *n* milliseconds.
    - Return immediately.
    - Temporarily cache a message.

3.5.3 Windows: Advanced local procedure call

- Only works between processes on the same system.
- Uses mailbox-like ports for communication channels.
- Works as follows:
    - Client opens a handle to the subsystem’s connection port object.
    - Client sends a connection request.
    - Server creates two private communication ports, returns the handle to one of them to the client.
    - Client and server use the comm. Port to send messages/callbacks and listen to replies.

3.6 Communications in Client-Server Systems

Sockets

- Connection between two processes, client and server. Two sockets needed in total.
- IP address and port make up the socket address.
- Ports below 1024 are well known, for example, port 80 is HTTP.
- Common and efficient, but not structured: data interpretation is handled by software.
- IP address 127.0.0.1 (loopback), refers to the same machine.

Remote procedure calls

- Abstraction of procedure calls between processes on networks.
- Uses ports for differentiation.
- Stubs: client-side proxy for the actual procedure on the server.
    - Stub locates the server, sends the parameters (specific term is “marshal”).
    - Server receives parameters, it un-marshals the parameters and performs the procedures on the server.
- Windows: stubs are created using the Microsoft Interface Definition Language (MIDL).
- Data representation handled with External Data Representation (XDL), accounts for different architectures (big endian and little endian).
- Problematic, has more failure scenarios than local execution, because of network errors some calls can be duplicated.
    - Solution? Ensure exactly *one* execution, rather than *at most* one.
        - At most once: all calls come with a timestamp attached, the server has a history log to detect repeated calls.
        - Exactly once: implement at most once, but include acknowledgement code (ACKs) between the server and the client, so that both parties are aware that the call has been executed.
- OS typically provides a matchmaker to handle the connections.

Pipes

- Conduit between two processes to communicate.
- Problems:
    - Communication goes one-way or two-way?
    - If two-way, half or full-duplex?
    - Is a parent-child relationship between the communication needed?
    - Can the pipes be used over network?

Ordinary pipes

- One process writes (write-end), the other process reads (read-end).
- One-way pipes.
- They need a parent-child relationship between the processes.
- Windows calls them *anonymous pipes*.
- Unix: array of two ints is used to keep track of the codes for the read/write pipes (see code at bottom of document).

Named pipes

- Two-way pipes.
- No parent-child relationship needed.
- Several processes can use the named pipe, it’s typical to have various writers to the same pipe.
- Unix: created with the `mkfifo()` syscall.

3.7 Answers of question 3.1 to 3.7

3.1 Using the program shown in Figure 3.30, explain what the output will be at LINE A.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int value = 5;
int main()
{
      pid_t pid;
      pid = fork();
      if (pid == 0) { /* child process */
            value += 15;
            return 0;
      }
      else if (pid > 0) { /* parent process */
            wait(NULL);
            printf("PARENT: value = %d",value); /* LINE A */
            return 0;
      }
}
```

Since the copy of the child process is not shared with the parent, LINE A will print 5, the parent’s original value.

3.2 Including the initial parent process, how many processes are created by the program shown in Figure 3.31?

```
#include <stdio.h>
#include <unistd.h>
int main()
{
      /* fork a child process */
      fork();
      /* fork another child process */
      fork();
      /* and fork another */
      fork();
      return 0;
}
```

Parent

- Child 1
    - Child 1.1
        - Child 1.1.1
    - Child 1.2
- Child 2
    - Child 2.1
- Child 3

8 processes in total.

3.3 Original versions of Apple's mobile iOS operating system provided no means of concurrent processing. Discuss three major complications that concurrent processing adds to an operating system.

- OS has to keep track on the memory space address so that no process touches the memory space of another process.
- Switching in-and-out processes consumes time, you have to save stuff on the PCB, and it creates overhead that has to be considered for efficiency.
- Swapping from disk to memory becomes a problem, as you may need to move some processes that no longer fit in memory back onto a disk to make space for the process that's exiting the CPU.

3.4 The Sun UltraSPARC processor has multiple register sets. Describe what happens when a context switch occurs if the new context is already loaded into one of the register sets. What happens if the new context is in memory rather than in a register set and all the register sets are in use?

- If the to-be-loaded context is in memory, simply changing the current-register-set pointer to point to said context is simple and cheap in term of operations.
- If a new context has to be loaded from memory and all slots are taken, then one of the loaded contexts has to be unloaded onto memory to load the new one. Overhead is created by this.

3.5 When a process creates a new process using the fork() operation, which of the following states is shared between the parent process and the child process?

a. Stack

b. Heap

c. Shared memory segments

Stack and heap are created for the child process, the shared memory segments are, however, shared between the processes.

3.6 Consider the "exactly once" semantic with respect to the RPC mechanism. Does the algorithm for implementing this semantic execute correctly even if the ACK message sent back to the client is lost due to a network problem? Describe the sequence of messages, and discuss whether "exactly once" is still preserved.

If the ACK message sent back to the client is lost, a good implementation of RPC would have the client have a *timeout*. If the timeout is reached after sending the request and not receiving the ACK, it thinks that the server did not (for whatever reason) execute the call, and sends the request again.

The server, however, may have executed said call, and the original call is logged on the history log. Whenever the new request enters, it is checked against this log, and because it's the same call, the server ***must*** return an ACK to the client.

3.7 Assume that a distributed system is susceptible to server failure. What mechanisms would be required to guarantee the “exactly once” semantic for execution of RPCs?

The history log must be preserved *by all means*, preferably in disk, so that in case the server crashes, the log is not lost and repeated messages are still filtered out.

3.8. Code
ORDINARY PIPE ON UNIX

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

#define BUFFER SIZE 25
#define READ_END 0
#define WRITE_END 1

int main(void) {
     char write_msg[BUFFER SIZE] = "Greetings";
     char read_msg[BUFFER SIZE];
     int fd[2];
     pid_t pid;


     /* create the pipe */
     if (pipe(fd) == -1) {
          fprintf(stderr,"Pipe failed");
          return 1;
     }

     /* fork a child process */
     pid = fork();
     if (pid < 0) {
          /* error occurred */
          fprintf(stderr, "Fork Failed");
          return 1;
     }

     if (pid > 0) {
          /* parent process */
```

```
        /* close the unused end of the pipe */
        close(fd[READ_END]);

        /* write to the pipe */
        write(fd[WRITE_END], write_msg, strlen(write_msg) + 1);

        /* close the write end of the pipe */
        close(fd[WRITE_END]);
    } else {
        /* child process */
        /* close the unused end of the pipe */
        close(fd[WRITE_END]);

        /* read from the pipe */
        read(fd[READ_END], read_msg, BUFFER SIZE);
        printf("read %s", read_msg);

        /* close the write end of the pipe */
        close(fd[READ_END]);
    }

    return 0;
}
```