Operating Systems

Process

Carlos Alberto Llano R.

6 de agosto de 2015

Contents

What is a process?

A process is a program in execution;at any instant in time.

- A program by itself is not a process. A program is a passive entity.
- A program becomes a process when an executable file is loaded into memory
- A process is an active entity.
- Components:
  - Memory: Instructions and data lie in memory
  - Registers: For example, the program counter(PC)
  - Other information as files and devices

Some concepts

- Time sharing
- Context switch
- Policies
- Scheduling policy

Time sharing

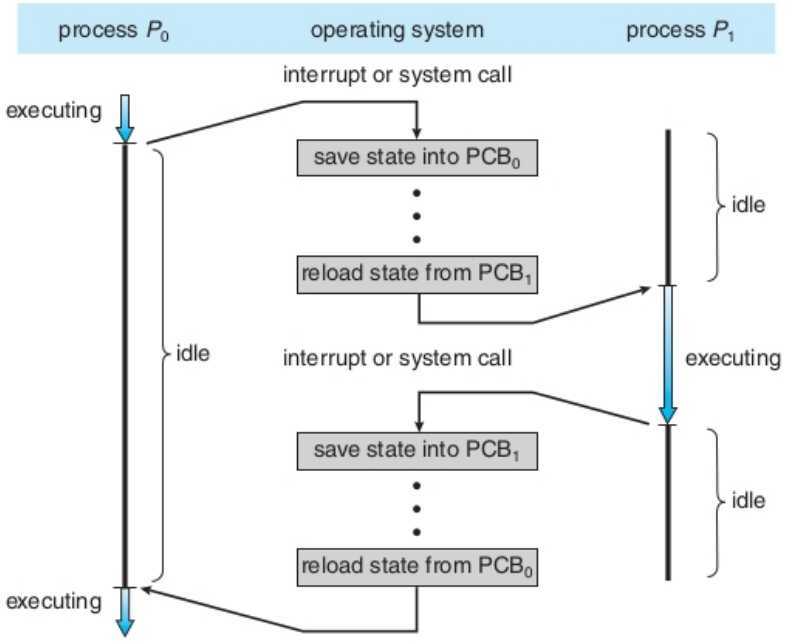
The OS creates this illusion by virtualizing the CPU. By running one process, then stopping it and running another, and so forth, the OS can promote the illusion that many virtual CPUs exist when in fact there is only one physical CPU (or a few).

Time sharing is one of the most basic techniques used by an OS to share a resource. By allowing the resource to be used for a little while by one entity, and then a little while by another, and so forth, the resource in question (e.g., the CPU, or a network link) can be shared by many.

Context switch

Context switch is the ability to stop running one program and start running another on a given CPU.

When an interrupt occurs, the system needs to save the current context of the process running on the CPU so that it can restore that context when its processing is done, essentially suspending the process and then resuming it.



Policies

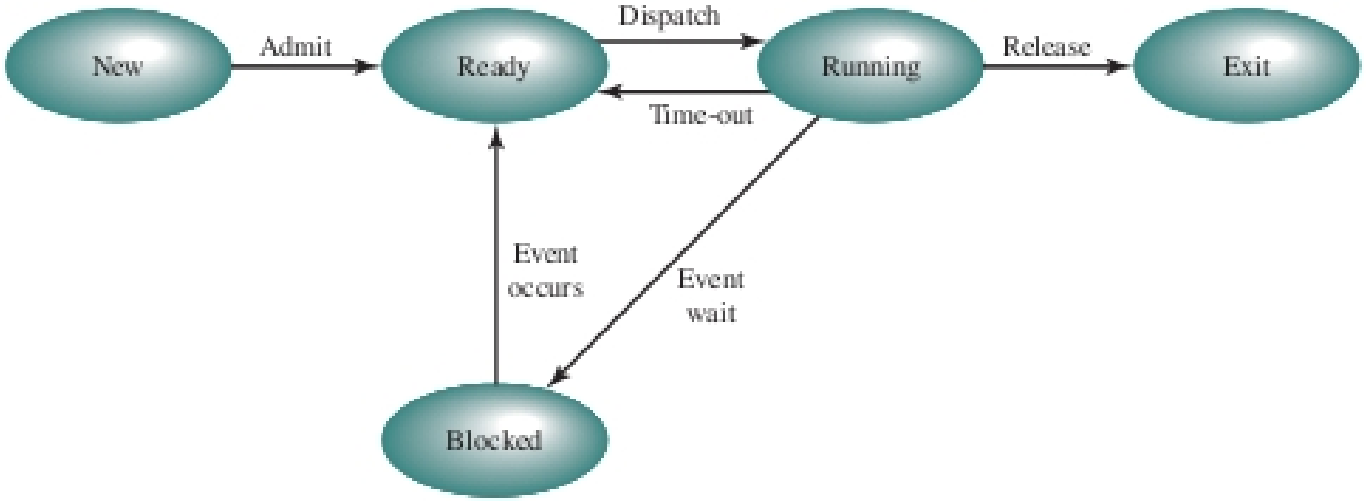
Policies are algorithms for making some kind of decision within the OS. For example, given a number of possible programs to run on a CPU, which program should the OS run?

Scheduling policy

Scheduling policy in the OS will make this decision, likely using historical information (e.g., which program has run more over the last minute?), workload knowledge (e.g., what types of programs are run), and performance metrics (e.g., is the system optimizing for interactive performance, or throughput?) to make its decision.

Process states

- New
- Running
- Waiting (or blocked)
- Ready
- Terminated



UNIX Process states

User Running	Executing in user mode.
Kernel Running	Executing in kernel mode.
Ready to Run, in Memory	Ready to run as soon as the kernel schedules it.
Asleep in Memory	Unable to execute until an event occurs; process is in main memory (a blocked state).
Ready to Run, Swapped	Process is ready to run, but the swapper must swap the process into main memory before the kernel can schedule it to execute.
Sleeping, Swapped	The process is awaiting an event and has been swapped to secondary storage (a blocked state).
Preempted	Process is returning from kernel to user mode, but the kernel preempts it and does a process switch to schedule another process.
Created	Process is newly created and not yet ready to run.
Zombie	Process no longer exists, but it leaves a record for its parent process to collect.

Tracing Process State: CPU and I/O

Time	Process_0	Process_1	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	Process_0 initiates I/O
4	Blocked	Running	Process_0 is blocked,
5	Blocked	Running	so Process_1 runs
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	Process_1 now done
9	Running	–	
10	Running	–	Process_0 now done

Reasons for process terminations

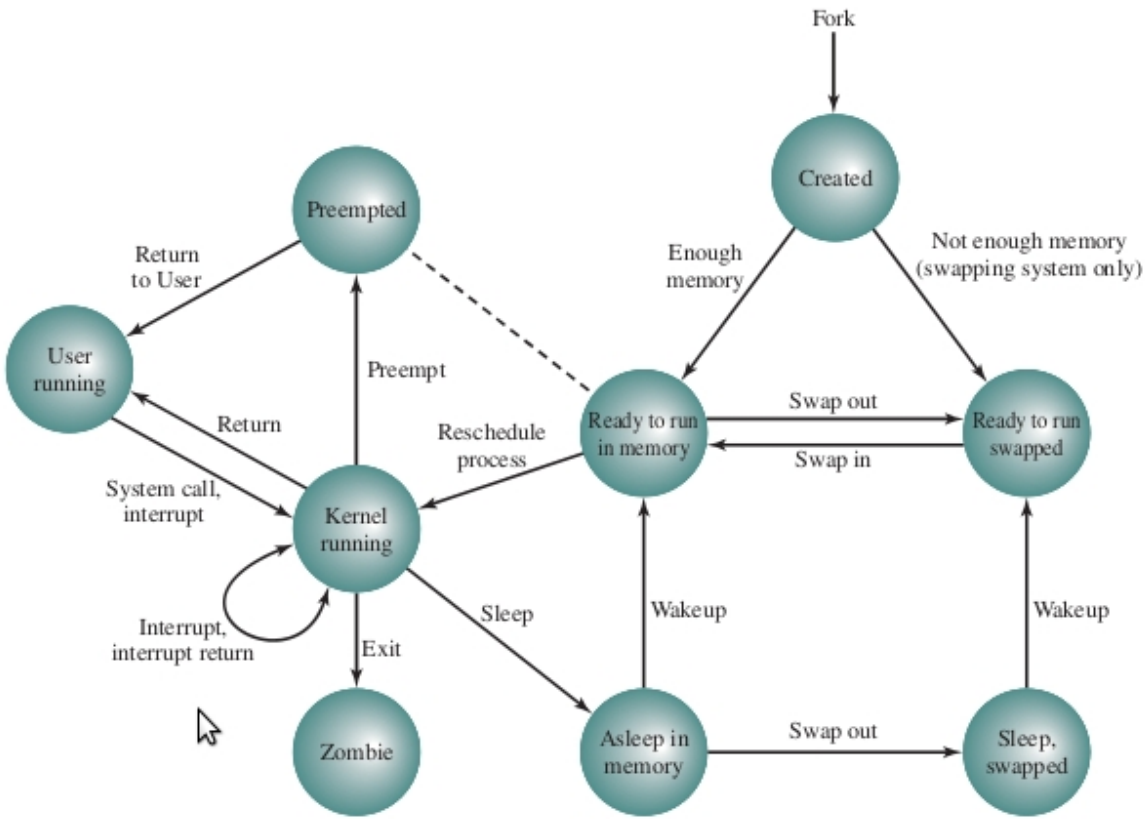
Normal completion	The process executes an OS service call to indicate that it has completed running.
Time limit exceeded	The process has run longer than the specified total time limit. There are a number of possibilities for the type of time that is measured. These include total elapsed time ("wall clock time"), amount of time spent executing, and, in the case of an interactive process, the amount of time since the user last provided any input.
Memory unavailable	The process requires more memory than the system can provide.
Bounds violation	The process tries to access a memory location that it is not allowed to access.
Protection error	The process attempts to use a resource such as a file that it is not allowed to use, or it tries to use it in an improper fashion, such as writing to a read-only file.
Arithmetic error	The process tries a prohibited computation, such as division by zero, or tries to store numbers larger than the hardware can accommodate.
Time overrun	The process has waited longer than a specified maximum for a certain event to occur.
I/O failure	An error occurs during input or output, such as inability to find a file, failure to read or write after a specified maximum number of tries (when, for example, a defective area is encountered on a tape), or invalid operation (such as reading from the line printer).
Invalid instruction	The process attempts to execute a nonexistent instruction (often a result of branching into a data area and attempting to execute the data).
Privileged instruction	The process attempts to use an instruction reserved for the operating system.
Data misuse	A piece of data is of the wrong type or is not initialized.
Operator or OS intervention	For some reason, the operator or the operating system has terminated the process (e.g., if a deadlock exists).
Parent termination	When a parent terminates, the operating system may automatically terminate all of the offspring of that parent.
Parent request	A parent process typically has the authority to terminate any of its offspring.

Swapping

To move part or all of process from main memory to disk. When none of the process in main memory is in the ready state, the OS swaps one of the blocked processes out on to disk into a suspend queue.

Swapping is an I/O operation.

UNIX Process State Transition Diagram

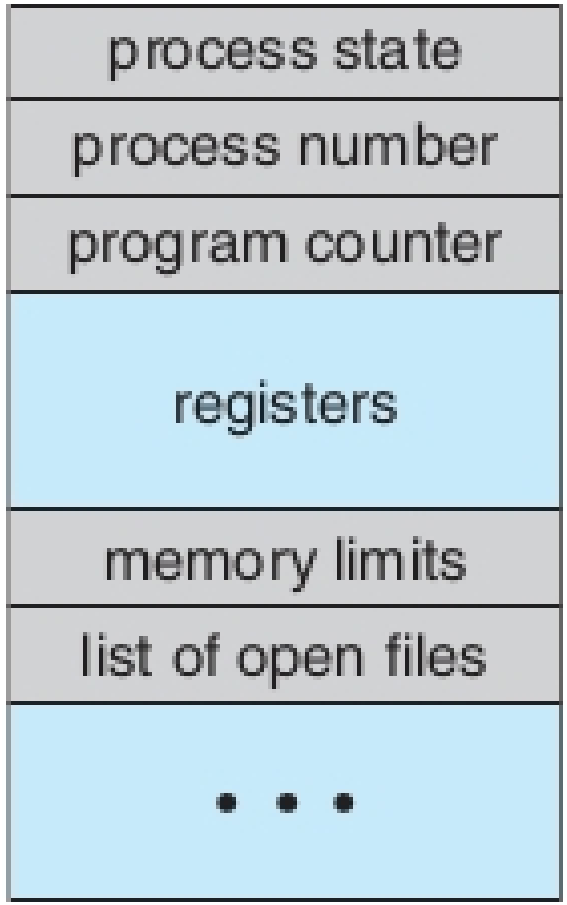


Process Control Block

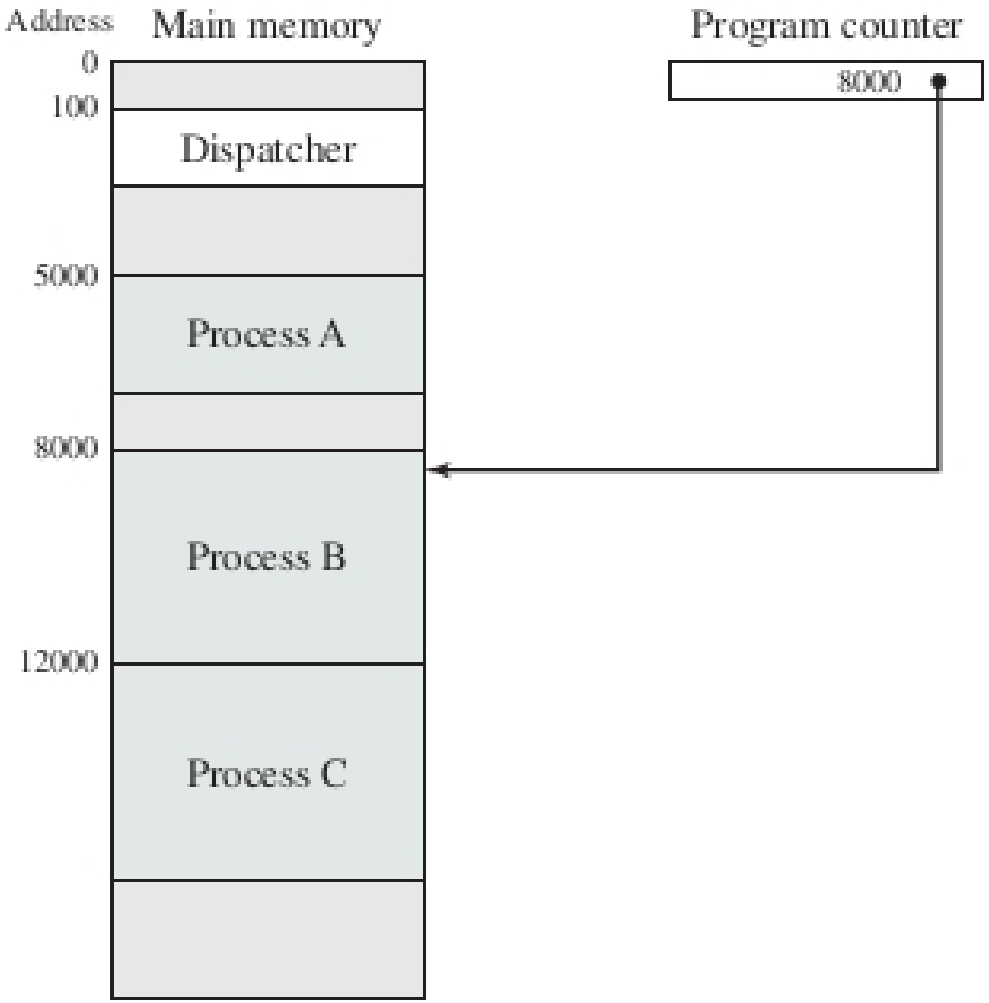
In brief, the PCB simply serves as the repository for any information that may vary from process to process.

- Process state
- Program counter: Address of the next instruction to be executed for this process.
- CPU registers: Depends of the computer architecture (Accumulators, index registers, stack pointers, etc.)
- CPU-scheduling information
- Memory-management information
- Accounting information: includes the amount of CPU and real time used and so on.
- I/O status information: includes the list of I/O devices allocated to the process (open files and so on)

Process Control Block

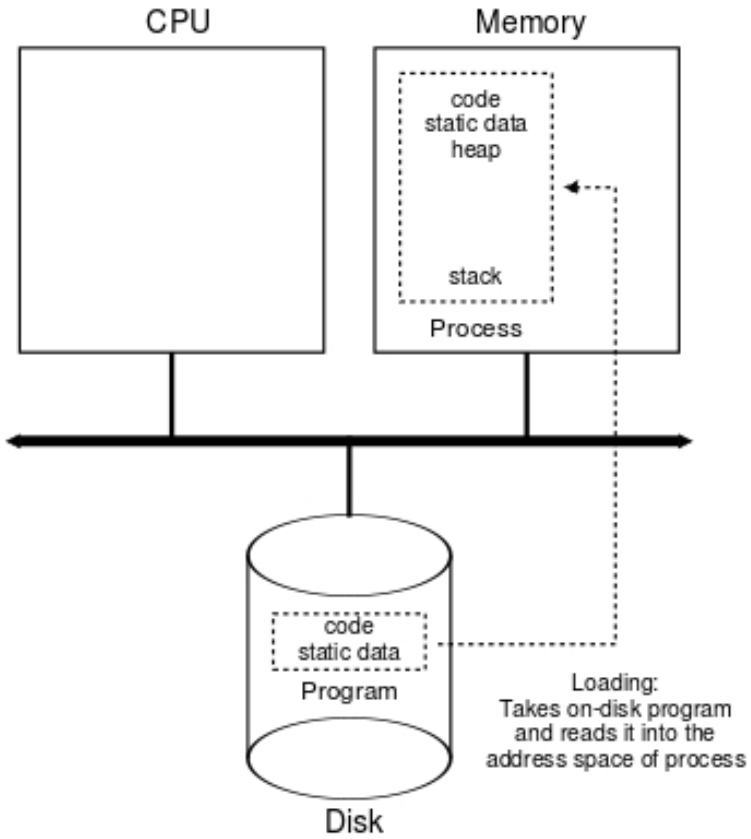


Program counter



Process creation

The first thing that the OS must do to run a program is to load its code and any static data (e.g., initialized variables) into memory, into the address space of the process. Programs initially reside on disk.



Process creation

Once the code and static data are loaded into memory:

- Some memory must be allocated for the program's run-time stack:
  - C programs use the stack for local variables, function parameters, and return addresses.
- Also allocate some memory for the program's heap (explicitly requested dynamically-allocated data, with malloc)

Process Structure in Linux

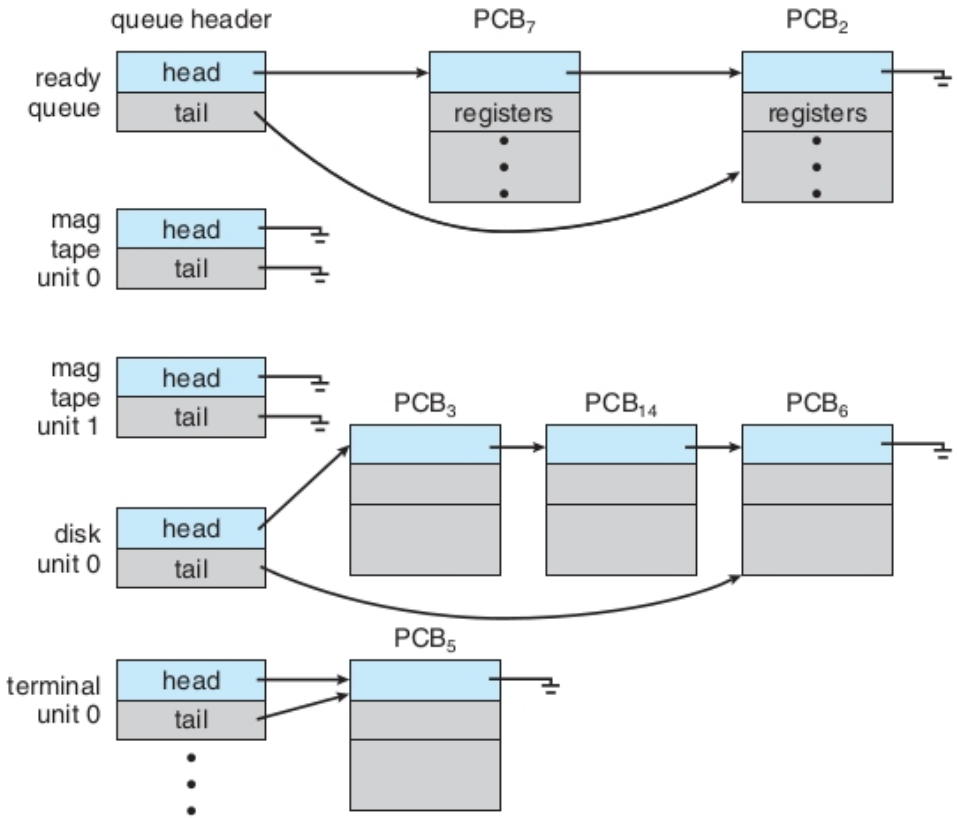
```
// the registers xv6 will save and restore
// to stop and subsequently restart a process
struct context {
  int eip;
  int esp;
  int ebx;
  int ecx;
  int edx;
  int esi;
  int edi;
  int ebp;
};

// the different states a process can be in
enum proc_state { UNUSED, EMBRYO, SLEEPING,
                  RUNNABLE, RUNNING, ZOMBIE };

// the information xv6 tracks about each process
// including its register context and state
struct proc {
  char *mem;                  // Start of process memory
  uint sz;                    // Size of process memory
  char *kstack;               // Bottom of kernel stack
                              // for this process
  enum proc_state state;      // Process state
  int pid;                    // Process ID
  struct proc *parent;        // Parent process
  void *chan;                 // If non-zero, sleeping on chan
  int killed;                 // If non-zero, have been killed
  struct file *ofile[NOFILE]; // Open files
  struct inode *cwd;          // Current directory
  struct context context;     // Switch here to run process
  struct trapframe *tf;       // Trap frame for the
                              // current interrupt
};
```

Threads

A process is a program that performs a single thread of execution.
Most modern operating systems have extended the process concept to allow a process to have multiple threads of execution and thus to perform more than one task at a time.
On a system that supports threads, the PCB is expanded to include information for each thread.

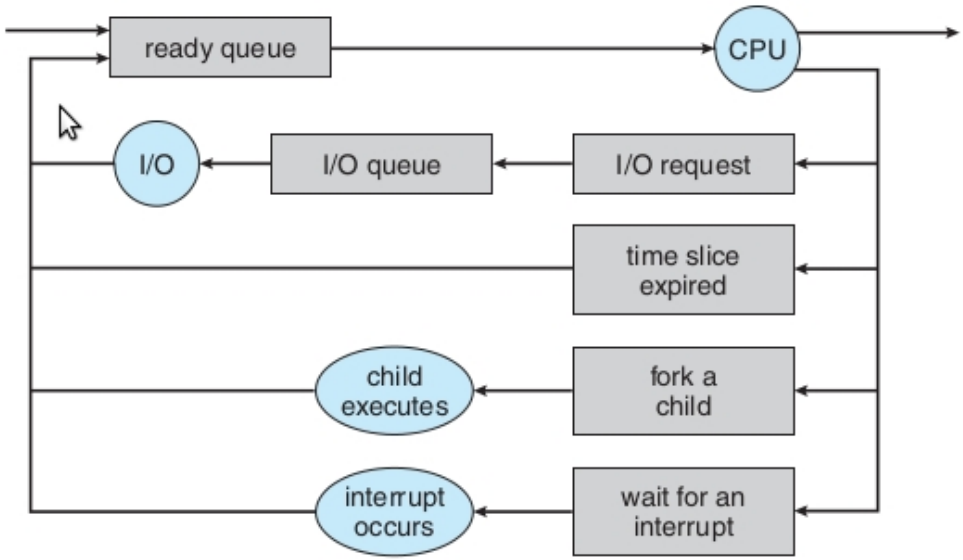
Process Scheduling

The process scheduler selects an available process (possibly from a set of several available processes) for program execution on the CPU.



For a single-processor system, there will never be more than one

Process Scheduling



Process API

- fork(): It is used to create a new process.
- wait(): It is quite useful for a parent to wait for a child process to finish what it has been doing.
- exec(): This system call is useful when you want to run a program that is different from the calling program.

Process API - Examples

Examples

Exercises

- Write a program that calls fork(). Before calling fork(), have the main process access a variable (e.g., x) and set its value to something (e.g., 100). What value is the variable in the child process? What happens to the variable when both the child and parent change the value of x?
- Write a program that calls fork() and then calls some form of exec() to run the program /bin/ls. See if you can try all of the variants of exec(), including execl(), execle(), execlp(), execv(), execvp(), and execvP(). Why do you think there are so many variants of the same basic call?

Interprocess Communication

Processes within a system may be independent or cooperating. A process is independent if it cannot affect or be affected by the other processes executing in the system.

Reasons for cooperating processes:

- Information sharing
- Computation speedup
- Modularity
- Convenience

Interprocess Communication

There are two fundamental models of interprocess communication:

- shared memory
- message passing

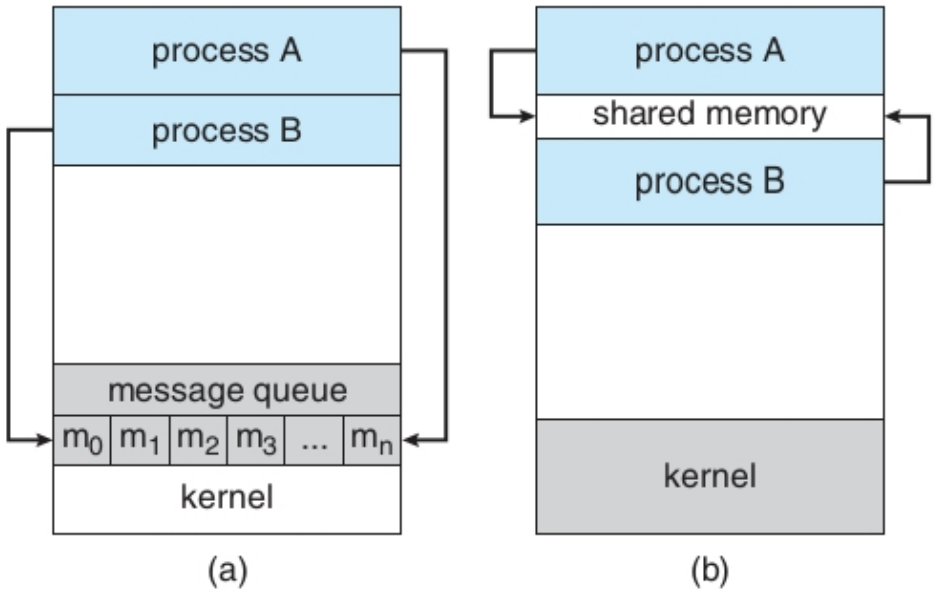
Shared-memory

In the shared-memory model, a region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region.

Message-passing

In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes.

Communications models



Producer–Consumer problem (Server-Client)

A producer process produces information that is consumed by a consumer process.

The idea is: To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer.

Producer–Consumer problem (Server-Client)

Two types of buffers can be used:

- The unbounded buffer: The consumer may have to wait for new items, but the producer can always produce new items.
- The bounded buffer: the consumer must wait if the buffer is empty.

Process Shared Memory

The buffer is empty when in == out the buffer is full when ((in + 1) % BUFFER SIZE) == out

```
#define BUFFER_SIZE 10

typedef struct {
    . . .
}item;

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

```
while (true) {
     /* produce an item in next_produced */

     while (((in + 1) % BUFFER_SIZE) == out)
        ; /* do nothing */

     buffer[in] = next_produced;
     in = (in + 1) % BUFFER_SIZE;
}
```

```
while (true) {
     while (in == out)
        ; /* do nothing */

     next_consumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;

     /* consume the item in next_consumed */
}
```

Process Shared Memory

Examples

Message-passing

Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space.

It is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network.

Message-passing

- Pipe: Chain of processes so that output of one process (stdout) is fed an input (stdin) to another.
- Message Queues: Two (or more) processes can exchange information via access to a common system message queue.
- Sockets: Sockets provide point-to-point, two-way communication between two processes.

Message-passing Linux pipe-examples

- grep processor /proc/cpuinfo | wc -l
- bunzip2 -c somefile.tar.bz2 | tar -xvf -
- grep -rl "old" ./ | xargs sed -i "s/old/new/g"
- tar zcpf - wsm | ssh root@192.168.10.112 "tar zxpf - -C /root"
- ifconfig -a | grep eth
- grep ïnterface/var/vshell/etc/shell.conf | cut -d'=' -f 2 | tr -d '[[:space:]]'

Message-passing popen, message and sockets examples

Examples

msgget - Warning

System-wide limit on the number of message queues. Before Linux 3.19, the default value for this limit was calculated using a formula based on available system memory. Since Linux 3.19, the default value is 32,000. On Linux, this limit can be read and modified via /proc/sys/kernel/msgmni.

Exercises

- To improve the code socket_server.c for the server remains listening more requests.
- Write a 2 programs that will both send and messages and construct the following dialog between them
  - (Process 1) Sends the message "Are you hearing me?"
  - (Process 2) Receives the message and replies "Loud and Clear".
  - (Process 1) Receives the reply and then says "I can hear you too".
- Investigate and understand fully the operations of the flags (access, creation etc. permissions) you can set interactively in the programs.
- Write a server program and two client programs so that the server can communicate privately to each client individually via a single message queue.