CS2106

Introduction to Operating Systems

AY2022/23 Semester 2

Notes by Jonathan Tay

Last updated on February 2, 2023

Contents

Part I

Process Management

1 Process Abstraction

A **process** abstracts the information required to describe and manage the execution of a program.

1.1 Function Invocation

Function invocation presents control flow and data storage challenges:

- jumping to function body, and resuming after the function returns
- passing parameters, and capturing a return value
- allocating local variables

stack memory

Stack memory is a region of memory separate from instruction and data memory that is used to store the following information for function invocations in **stack frames**:

1. return address (PC) of the caller
2. arguments for the parameters of the function
3. local variables used by the function
4. saved **stack pointer (SP)**

Optionally, the stack frame also includes:

5. **frame pointer (FP)**
6. saved registers

Stack frames are **pushed** onto the stack on invocation and **popped** on return.

The **stack pointer (SP)** is a special register which contains the address of the first free location in the stack.

Because SP varies depending on the number of local variables, some processors provide a **frame pointer (FP)** which points to a fixed location in a stack frame.

When all general purpose registers are exhausted (**register spilling**), their values can be written to memory and restored at the end of the function call.

function invocation — setup

If a function `f()` invokes `g()`, then `f()` is the **caller** and `g()` is the **callee**.

- **caller**:
  1. passes arguments by setting registers directly and/or pushes arguments onto the stack
  2. saves the return address on the stack
- **callee**:
  1. saves the old stack pointer
  2. allocates space for local variables
  3. sets SP to the top of the stack

function invocation — teardown

- **callee**:
  1. pushes return value onto the stack (if any)
  2. restores SP to the saved stack pointer
  3. sets PC to the return address
- **caller**:
  1. uses the return result (if any)
  2. resumes execution

Actual implementation of the function call convention depends on hardware, programming language, and compiler — it is not universal.

1.2 Dynamically Allocated Memory

Some data may have a size only known at runtime, which precludes allocation in the data memory.

Some data may have an unknown lifetime (i.e. no deallocation time guarantees) which precludes allocation in the stack memory.

Hence, such dynamic data is allocated (e.g. by `malloc`) in the **heap memory**.

1.3 Process Model

Process IDs (PIDs) uniquely identify processes, each of which may exist in 5 different possible states:

1. **new**: created, or under initialization
2. **ready**: admitted, but waiting to run
3. **running**: being executed by the CPU
4. **blocked**: waiting until an event occurs
5. **terminated**: finished execution, possibly requiring OS cleanup

process control blocks (PCBs)

Every process' registers, memory, PID, and state is stored in a **process control block**.

The kernel manages all PCBs in a **process table**.

context switching

Programs may be interrupted at any time, e.g. by the **scheduler**, and must operate independently of interrupts.

A **context switch** saves the current execution context process in a PCB and loads the context of the next process to be executed.

1.4 System Calls

System calls, or syscalls, are an API for programs (or user-friendly libraries) to call services in the kernel.

Each syscall has a unique **system call number**, which is passed to the kernel via a designated register along with any arguments.

The signature of a syscall in C is `long syscall(long number[, arg1, arg2, ...])`.

These functions require a change from user mode to kernel mode when invoked, via a **TRAP instruction**.

Syscalls are laid out in a table, which a **dispatcher** uses the system call number to find the correct **system call handler**.

The system call handler executes the request, returns control to the library call, and switches back to user mode.

1.5 Exceptions and Interrupts

Machine level instructions can cause ***synchronous* exceptions** during program execution, e.g. division by zero or illegal memory access, which then passes control to an **exception handler**.

External events such as timers or hardware can cause ***asynchronous* interrupts**, which the operating then searches the **interrupt vector table** (IVT) to invokes the correct **interrupt handler**.

The IVT is set up by the OS during boot, and exists in memory (hardware).

As interrupts are asynchronous, they are only checked at specific times; never in the middle of an instruction: instructions are atomic/indivisible.

exception/interrupt handlers

These handlers save the register and CPU state, handle the necessary routine, and then restore the state and return to the interrupted program.

Program execution may resume as though the exception or interrupt never occurred.

1.6 Example: Unix Processes

Every process has an integer process ID (PID). Child processes also remember their parent's PID.

`init` is the ancestor of all other processes, created by the kernel at boot, with PID 1.

Process status can be viewed with the `ps` command.

`fork()`

`fork()` creates a new **child process** which shares the same code and address space as the parent process.

Only the data memory (e.g. variables) of the child process is copied from the parent instead of shared.

Because memory copying is expensive, some systems use **copy-on-write**.

`fork()` returns `0` from the child process and the PID of the child process from the parent process.

`execl()`

`execl()` replaces the current process with a new process specified by the first argument, with optional `NULL`-terminated command-line arguments:

```
int execl(char *path, char *arg, ..., NULL);
```

wait()

`wait()` blocks the parent process until at least one child process terminates, after which it clears the remainder of the child process' resources which are not removed by `exit()`.

`wait()` returns the PID of the terminated child process, while also accepting an optional `int` pointer for the child process' exit status:

```
int wait(int *status);
```

If the parent process terminates before its child processes, the **orphan** child processes are adopted by `init` which will use `wait()` to cleanup.

If the child process terminates with `exit()` but its parent does not call `wait()`, then it becomes a **zombie process**.

2 Inter-process Communication

Two processes P_1 and P_2 need to communicate with each other. There are two ways to do this:

1. **shared memory**:
   P_1 creates a shared memory region which P_2 can attach to. P_1 and P_2 can then read and write to the shared memory region.
2. **message passing**:
   P_1 and P_2 exchange messages, where sending and receiving involve the OS via syscalls.

The OS overhead for message passing is much higher than for shared memory, where the OS is only involved during the setup and teardown.

Both options require some form of synchronization, to avoid **data races**, where the processes read and write to the same memory location at the same time.

Race conditions may result in incorrect behavior.

direct & indirect communication

In a shared memory scheme, communication is always *implicit* — a bystander cannot easily detect that communication is taking place.

However, in message passing, communication is *explicit* — the sender must explicitly name the recipient, or specify a common message storage, usually a shared **mailbox** or **port**.

This means message passing may require the use of finite *kernel memory* to buffer messages.

synchronous & asynchronous message passing

The `receive()` operation is always assumed to be *blocking* — always waiting until a message arrives.

Asynchronous message passing is *non-blocking*, requiring the kernel to buffer messages. In the case of buffer overflow, `send()` may:

- block until the buffer has space, or,
- return an error.

Synchronous message passing is *blocking*, requiring the sender to wait until the receiver has received the message. No buffering is required.

2.1 Example: Unix IPC

shared memory

1. The master program creates the shared memory space via `shmget()` and attaches it via `shmat()`.
2. The worker process only needs to attach to the shared memory space via `shmat()`.
3. Both processes use some space in the shared memory as control values for synchronization.
4. After work is complete, the worker process detaches from the shared memory via `shmdt()`.
5. The master process detaches from the shared memory via `shmdt()` and destroys the shared memory via `shmctl()`.

message passing — pipes

Every process has `stdin`, `stdout`, and `stderr` pipes, which can redirected with <, >, and 2> respectively.

Pipes in a **producer-consumer** relationship are connected with | at the command line.

Pipes are created with the `pipe(fd[2])` syscall in C, which returns two file descriptors, one for reading and one for writing.

The producer must `close()` the read end of the pipe before writing to it, and then `close()` the write end of the pipe after writing to it.

The consumer must `close()` the write end of the pipe before reading from it, and then `close()` the read end of the pipe after reading from it.

signals

Signals are asynchronous notifications sent to a process by the OS or another process.

Processes can define their own signal handlers to handle signals, such as `SIGSEGV`, using the `signal(SIGNAL, *fp)` syscall in C.

The `SIGKILL` and `SIGSTOP` signals are sent by the kernel to a process to terminate it, and cannot be caught, blocked, or ignored.