CS2106

Introduction to Operating Systems

AY2022/23 Semester 2

Notes by Jonathan Tay

Last updated on January 18, 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.
