Processes

Chapters: 3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.7, 3.8, (no 3.7.2, 3.7.3, 3.8.2.1), 4.6.2, 20.9.1, C3.3

# Overview

* **Process** is the OSs abstraction for **execution**
  + Program: list of instructions, initialised data etc,
  + Process is a **program in execution**
* **Sequential process**: single flow/sequence of instructions in memory
* **Address space**: abstraction of idea of memory. We are given a bunch of addresses that we can “put data in”. This abstracts the memory on the actual resource
* Only **one** process runs on a processor core at any time, but different processes may run the same program.
* A process consists of everything you need to run the program. This includes:
  + Address space containing code (instructions) and data (static data, heap data etc.) for running program
  + CPU state, consisting of:
    - program counter indicating next instruction
    - Stack pointer, current stack position (where are we in executing our functions)
    - Other general-purpose register values.
  + OS resources
    - Open files, network connections etc.

# Address Space

* Diagram

  Description automatically generated32- bit addresses means we have addresses
* The addresses abstract away the physical addresses of the memory.
* Each process has its own address space.
  + **Text section**: **Executable code**
  + **Data Section**: Global Variables
  + **Heap** section: Memory space for **dynamically allocated memory**
  + **Stack** section: temporary data store for invoking functions. Includes parameters, return addresses and local variables.

# OS Process Namespace

* Each process is identified by a **process ID** (**PID**), an integer
* PID namespace is **global** tosystem
  + Only one process at a time has a specific PID, exceptions do exist however (e.g. c(ontrol)groups)
* Operations that create processes return a PID e.g. fork()
* Operations on processes take PID as argument e.g, kill(), wait(), nice()

# Representation of Processes by OS

* To keep track a process’s state, we use a **process control block** (**PCB**) or **process/task descriptor**, identified by the PID and stored in RAM.
* OS keeps all of a process’s execution state in or linked from the PCB when the process isn’t running.
* When the process is running, its state is spread between the PCB and the CPU registers.
* Diagram, table

  Description automatically generatedPCB contains over a 100 different fields. This includes:
  + process ID (PID)
  + parent process ID
  + execution state
  + program counter, stack pointer, registers
  + address space info
  + UNIX user id, group id
  + scheduling priority
  + accounting info
  + pointers for state queues
* Note: multi-thread PCBs would hold more information

# PCB and CPU State

* When a process is running, its CPU state is inside the CPU. This includes the PC, SP, and registers.
* CPU will hold **current** values according to PC
* When OS obtains control via **syscall**, **exception** or **interrupt**, the OS saves the CPU state of the running process **in that’s process’s PCB**
* The OS executes and when the OS returns the process to the running state, it loads the hardware registers with values from that process’s PCB. This includes:
  + General purpose registers
  + stack pointer
  + instruction pointer
* **Context switch**: the act of switching a process from one to another
  + 100s-1000s switches/second made by computer systems, takes a few microseconds
  + Still expensive related to thread-based context switches
* Ordering these processes is **scheduling**

# Process context switch

* Diagram

  Description automatically generated**Interrupts** cause the OS to change a CPU core from its current task to run a kernel routine.
* **Context: PCB** for current process. Saved for post-processing continuation.
* Requires a lot of hardware support

# Process Execution States

* Each process has an **execution state**. These are:
  + Diagram

    Description automatically generated**New**: process is being created
  + **ready**: waiting to be assigned to CPI
  + **running**: executing on a CPI
  + **waiting** (blocked): waiting for an event e.g. I/O completion
  + **Terminated**: process has finished execution.
* Possible transitions between states shown by diagram

1. Diagram

   Description automatically generatedProcess is created. Is not running as it must be scheduled by OS
2. Once in running state, can either terminate (rare), pre-empt (interrupt itself and allow other processes to run) or go to waiting state for I/O
3. If gone to waiting, the I/O will run and after completion send an interrupt to the OS. This puts the process back in the ready state.

* Note that only scheduling can allow processes to run

# State Queues

* OS maintains collection of queues, representing state of all processes in system.
* Typically there is one queue per state except from running, but there could also be multiple waiting queues.
* Each PCB is either running, or is exclusively in one these queues according to its state.
* As processes change state, its PCB is **unlinked from** one queue and **linked onto** another**.**

Diagram

Description automatically generated

* There may be many wait queues, one for each type of wait (e.g. specific device, timer, message, etc.)
* PCBs are **data structures**, and are **dynamically allocated** inside OS memory.
* When a process is **created**:
  + OS allocates a PCB for it.
  + OS initialises PCB values.
  + (OS does other things such as initialise additional management states)
  + OS places PCB on the correct queue.
* As the process **computes**:
  + OS moves PCB from queue to queue
* When process is **terminated**:
  + PCB may be retained for a while (to receive signals etc.)
  + Eventually, OS deallocates the PCB

# Process Operations

## Process Creation

* New processes are created by existing processes
  + **Parent**: creator
  + **Child**: created process
  + **PPID**: parents ID field.
* List all processes: ps -el
* Diagram

  Description automatically generatedBoot instructions run the init process to start the tree of processes. Typically start services such as web or print servers, ssh servers etc.:
* Child processes can either:
  + Obtain resources directly from OS
  + Constrained to subset of resources from parent.
* On some systems, resource allocation to a parent may be divided among children. This prevents a process from overloading the system by creating too many child processes.
* In addition, the child process will inherit certain attributes such as privileges and scheduling permissions.
* Two possibilities for execution exist:
  + Parent process continues to execute concurrently with children
  + Parents wait until some or all of its children have terminated.
* In addition there are two address-space possibilities for a new process:
  + Child process is duplicate of parent process (same program and data)
  + Child process has a new program loaded into it.
* Timeline

  Description automatically generatedUNIX process creation achieved through fork() call
* This creates and initialises a new PCB
  + Initialising kernel resources of new process with resources of parent
  + Initialises PC and SP to be same as parent
* Creates a new address space
  + initialises new address space with **copy of entire contents** of the address space of parent
* Then PCB placed on ready queue.
* fork() call returns two outputs: child’s PID to parent and 0 to child.
* Note: Windows uses CreateProcess() instead, and has a few differences:
  + Requires loading a specified program into the address space instead of inheriting from parent.
  + Expects no fewer than 10 parameters.
* The two arguments returned by fork() make it unique. It returns two things:
  + the child ID to the parent process
  + 0 to the newly created process
* Diagram

  Description automatically generatedOnce fork() is called, it will execute from the point from which the fork() call was made

## Process Termination

* exit() syscall used by process once it is finished and ready to be deleted.
* Process may return status to a waiting parent process who has used wait()
  + wait() has an **optional** parameter that allows parents to obtain exit status of child process, and also returns the process ID of the terminated child.
    - (pid = wait(&status);
  + When a child is made, its ID and exit status is held in the process table.
  + The process table cannot release an entry until the parent calls wait() as otherwise the exit status would be lost.
  + A process that has terminated but whose parent has not invoked wait() is a **zombie process**.
  + A process who’s parent has terminated without invoking the wait() command is called an **orphan**.
    - To release an orphan, the init() process periodically invokes wait().
    - Since all processes are children of init(),then all exited processes will be released from the process table.
* All resources of process are deallocated and reclaimed by OS, including
  + Virtual and physical memory.
  + Open files.
  + I/O buffers.
* A parent can also kill its own child process. This can happen because of the following circumstances:
  + Child has exceeded of usage of (some of) its allocated resources. Parent must be able to inspect child’s state to check this).
  + Task assigned to child is no longer required
  + It the parent themselves are terminating (**cascading termination**)

## Android Process Hierarchy

1. **Foreground process**: visible on screen and user is interacting directly with it.
2. **Visible process**: Not directly visible but performing a task for a foreground process.
3. **Service process**: performing activity that is apparent to the user
4. **Background process**: performing an activity that is not apparent to the user.
5. **Empty process**: process that holds no active components associated with any application.

* Processes are terminated in reverse order.
* Rank of process is done based on maximum importance e.g. visible service process is classed as visible.
* State of terminated process can be saved for user in case they decide to back to the application.

# Inter-process Communication

* **Independent**: if process does not share data with any other processes executing in system.
* **Cooperating**: if process can be or affects other processes executing in the system.
* Separate independent processes communicating and cooperating. This involves:
  + **Information Sharing** if they are using the same information
  + If they are doing the same task, then **computation speedup** is essential
  + **Modularity** so that reusable components are carried out by other processes
* This requires an **inter-process communication** (**ICP**) mechanism to send and receive data.
  + This involves the use of **shared memory** and **message passing** between processes
* Diagram

  Description automatically generatedFrom the perspective of the OS, this is what shared memory and message passing looks like:
* Kernel, process A, process B: computer memory.
* Shared memory is an area of memory shared between several processes
* Note that the processes have independent address spaces i.e. they are invisible to one another.
* Shared memory is a special region that they can both use and hence use it to exchange data.
* Message passing uses enqueueing and dequeuing to send and receive messages from the OS.

## Shared Memory

* Allows processes to communicate and synchronise by sharing some address space.
* OS does not mediate between so there is **no OS overhead**
* Normally OS prevents processes from accessing each other’s memory, so processes need permission to void this restriction
* The data format for data transfer is decided by the application using said processes.
* Due to the direct access nature of shared memory it is **very fast**, but it is **not trivial**.
* Can be used for passing large (single) objects e.g. copy and paste, or for notification variables (see book)
* Works at a solution for **consumer-producer** problem using **bounded** or **unbounded** buffers.

## Message Passing

* Message passing: uses messages and OS to communicate with one another. This introduces overhead.
* Processes from same machine can use this.
* Also **among different networked machines** can use this to communicate: not possible with sharing memory
* Message-passing facility provides at least two operations
  + send(message)
  + receive(message)
* A **communication link** must be establishedi.e. how the messages are transferred.
* Link has different characteristics e.g. messages having fixed or variable size. Purely engineering difference.

### Naming

* Two types of communication: **direct** or **indirect**:

#### Direct Communication

* Splits into two types:
  + **Symmetric**: names of sender and receiver **explicitly named**:
    - send(P, message) – send message to process P
    - receive(Q, message) – receive message to process Q
  + **Asymmetric**: names of receiver only **explicitly named**:
    - send(P, message) – send to P
    - receive(ID, message) – receive message from any process, sender saved in ID.

* Disadvantages of both of these schemes is limited modularity e.g. changing process ID necessitates examining all other process definitions of the process.

#### Indirect Communication

* No need to know sender/receiver explicitly/in advance
* Messages are sent and received from **mailboxes** (**ports**).
  + Each mailbox has a unique ID e.g. POSIX message queues use integers to identify mailboxes.
  + Two processes can only communicate if they have a shared mailbox.
    - send(A, message) – send to mailbox A
    - receive(A, message) – receive message from mailbox A.

* Can be accessed by many processes and multiple mailboxes may exist.
* So what happens two or more processes issue a receive() command from a mailbox? 3 options:
  + Allow link to be associated with only 2 processes
  + Allow only one process at a time to execute receive()
  + System identifies which process receives message arbitrarily or through algorithm like round-robin
* Both OS and process can created mailboxes.
  + OS mailboxes are independent and not attached to any particular process.
  + Ownership and receiving privileges can be passed on via syscalls.

### Synchronisation

* Synchronising sending and receiving can come with different characteristics and challenges:
  + Different **design options** to implement send()/receive()
    - **Blocking**/**Synchronous**: blocks processor until message is received
    - **Non-Blocking**/**Asynchronous**: queues message passing for computation later.
  + Different **combinations** may be offered as well.
    - **Blocking send**
    - **Nonblocking send**
    - **Blocking receive**
    - **Nonblocking receive**
  + **Rendezvous**: when both send() and receive() are blocking. Simple solution to consumer-producer problem

#### Consumer-Producer Problem

* Fixed buffer size, producer process and consumer process
* Producer creates item and adds it to buffer. Consumer process “consumes” item from buffer
* To ensure consistent data synchronisation
  + Producer cannot produce item if buffer is full
  + Consumer cannot consume item if buffer is empty
  + Access to buffer must be **mutually exclusive** i.e. only one process accesses it at a time.

### Buffering

* Message exchanged between processes usually reside within **temporary buffers**/**queues**.
* Three different types:
  + Zero capacity (no buffering)
    - No message waiting in it
    - Sender must block until the recipient receives the message
  + Bounded capacity
    - Finite length n: at most n messages can reside in it
    - If the link is full, the sender has to wait
    - Most common
  + –Unbounded capacity
    - Infinite queue
    - The sender never blocks
    - Never found

### Example: Pipes

* Diagram

  Description automatically generated with low confidencePipes act as **conduit** allowing processes to communicate
  + One way data flow as shown
* Two types of pipes:
  + **Anonymous**: between any parent and child. half-duplex and unidirectional
  + **Named**: between any pair of processes (used in FIFO, UNIX). Full-duplex on windows and bidirectional

### Client-Server Communication

* Most famous example: **sockets** abstraction:
  + sockets are **endpoints for communication**
  + Identified by **IP address** concatenated with **port number**.
  + Servers implementing specific services such as SSH or HTTP listen to specific, well-known ports e.g. SSH listens to port 22.
* Another example: **Remote Procedure Call** (**RPC**)
  + Abstracts the procedure-call mechanism
  + Used for systems with network connections, using message based communication to provide remote services.

## Signals

### OS Signals

* OS mechanism to **notify a process**. This is one way and does not carry information
* Thought of as a software-generated interrupt/exception. Can be either:
  + Synchronous: generated by running process e.g. division by 0 error: instant
  + Asynchronous: generation external to running process e.g. operating on timer or alarm or CTRL+C command
* Syscall act from process to OS, signals act from OS to process
* **Signal handling** handled by:
  + **default** signal handler.
  + **User** signal handler (can override default handler)
* For **multithreaded** systems, signals are handled in one of four ways:
  + Signal delivered to threat to which it applies
  + Signal delivered to all threads
  + Signal delivered to specified threads.
  + Thread assigned for receiving all signals.

### Process Signals

* Two different processes can use signals for notifications: **no data transfer**
* Used in various ways:
  + Management e.g. kill a process
  + Synchronisation e.g. POSIX RT signals

### Using signals in Linux

* See lecture for more infoA picture containing text

  Description automatically generated.
* <https://www.cs.auckland.ac.nz/references/unix/digital/APS33DTE/DOCU_006.HTM#realtime-handler-sec>

### Realtime Signal Handling

* Traditional signals, as defined by POSIX 1003.1, have several limitations that make them unsuitable for realtime applications:
  + There are too few user-defined signals.
  + There are only two signals available for application use, SIGUSR1 and SIGUSR2. For those applications that are constructed from various general-purpose and special-purpose components, all executing concurrently, the same signal could trigger different actions, depending on the sender. To avoid the risk of calling the wrong signal handler, code must become more complex and avoid asynchronous, unpredictable signal delivery.
  + There is no priority ordering to the delivery of signals.
  + When multiple signals are pending to a process, the order in which they are delivered is undefined.
  + Blocked signals are lost.
  + A signal can be lost if it is not delivered immediately. A single bit in a signal set is set when a blocked signal arrives and is pending delivery to a process. When the signal is unblocked and delivered, this bit is cleared. While it is set, however, multiple instances of the same signal can arrive and be discarded.
  + The signal delivery carries no information that distinguishes the signal from others of the same type.
  + From the perspective of the receiving process, there is no information associated with signal delivery that explains where the signal came from or how it is different from other such signals it may receive.
* To overcome some of these limitations, POSIX 1003.1b extends the POSIX 1003.1 signal functionality to include the following facilitators for realtime signals:
  + A range of priority-ordered, application-specific signals from SIGRTMIN to SIGRTMAX
  + A mechanism for queuing signals for delivery to a process
  + A mechanism for providing additional information about a signal to the process to which it is delivered
  + Features that allow efficient signal delivery to a process when a POSIX 1003.1b timer expires, when a message arrives on an empty message queue, or when an asynchronous I/O operation completes
  + Functions that allow a process to respond more quickly to signal delivery