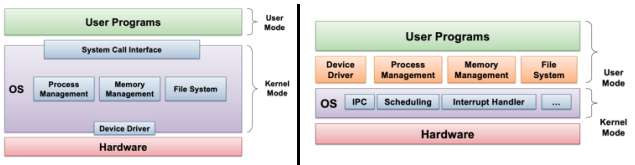
CS2106 Operating Systems

AY2021/22 Semester 2

1. Introduction

1.1. Introduction to Operating Systems

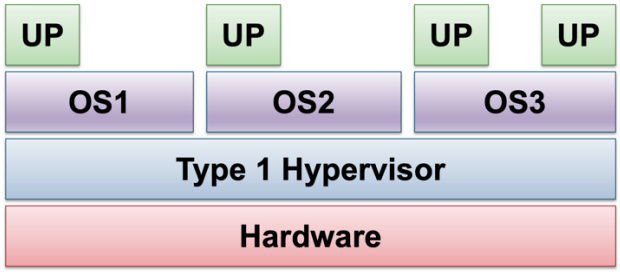
- OS is a program that acts as an intermediary between a computer user and the computer hardware.
- Monolithic VS Microkernel



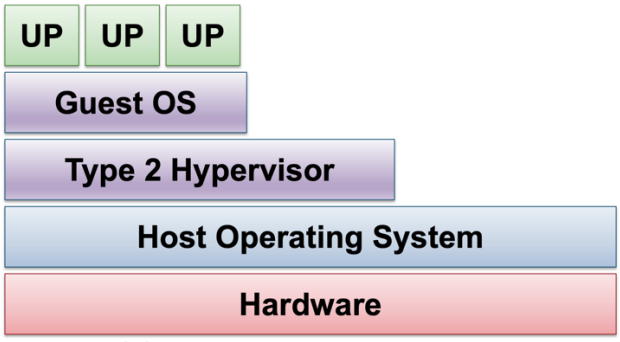
Pros: (1) Well understood (2) Good performance Cons: (1) Highly coupled components (2) Complicated internal structure	Pros: (1) Kernel is more robust and extendable. (2) Better isolation and protection between kernel and high-level services. Cons: Low performance

1.2. Virtual Machines (Hypervisors)

- Type 1 Hypervisor: Provides individual virtual machines to guest OSes.



  - Faster than Type 2 due to fewer overheads (one less layer)
- Type 2 Hypervisor: Runs in host OS.



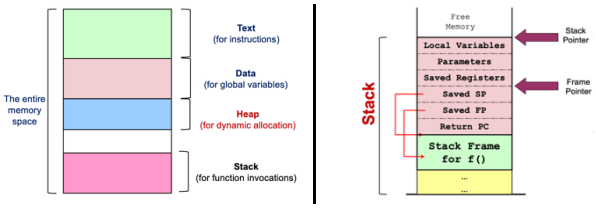
  - Simpler to build than Type 1

2. Process Management

2.1. Process Abstraction

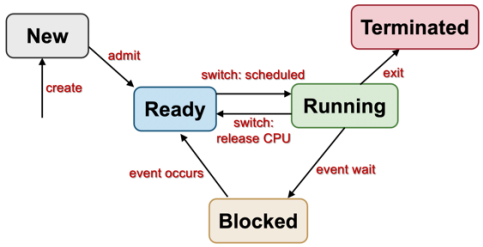
- Process is a dynamic abstraction for executing programmes. It includes all information required to describe a running programme.

- Memory Regions



  - Global Variable → Data Section
  - Function Variable → Stack Section
  - `malloc()` → Heap Section; The pointer itself is in the stack section.

- Generic 5-State Process Model



  - New: New process created.
  - Ready: Process is waiting to run.
  - Running: Process is being executed on CPU.
  - Blocked: Process is waiting for event.
  - Termination: Process has finished execution.
  - Block – Ready – Running

- System Calls: Changes from user mode to kernel mode (made by a library call).
  - Hides the lower level hardware details from user program.
  - Protects system integrity from user program.
  - Requires execution mode change and possibly context change, introducing overheads.

- Process Creation in UNIX – `fork()`
  - Returns PID for parent process and 0 for child process.
  - Child process is a duplicate of parent process. Parent process and child process have independent memory space.
  - exec(): Replaces current executing process image with a new one.
    - Code replacement
    - PID and other information still intact
  - `fork()` + `exec()`: Spawns off a child process and get parent process ready to accept another request.
  - Root process: `init`

- Process Termination in UNIX – `exit()`
  - Returns status to parent process.

- Unix convention: `0` for normal termination, `!0` for problems.
- Releases most system resources except PID, status, process accounting info, etc.

- Parent/Child Synchronisation in UNIX – `wait()`
  - Returns the PID of the terminated child process.
  - Blocks parent process until at least one child process terminates.
  - If there is no child, continue immediately.
  - Cleans up the remainder of child system resources and kills zombie processes.

2.2. Process Scheduling

- Evaluating Criteria
  - Fairness: Ensures fair sharing of CPU time, no starvation.
  - Balanced utilisation of system resources

- Non-preemptive VS Preemptive (CPU can be taken from Running state at any time)

- Scheduling Algorithms for Batch Processing
  - Batch processing: Long running without user intervention. Non-preemptive scheduling is dominant.
  - Criteria: (1) Throughput: Number of tasks finished per unit time, (2) Turnaround time: Total wall clock time taken, (3) CPU utilisation: Related to overheads.
  - First-Come First-Served (FCFS): Guaranteed to have no starvation.
    - FCFS minimises the average response time if the jobs arrive in the ready queue in order of increasing job lengths.
  - Shortest Job First (SJF): Minimises average waiting time but may cause starvation – long jobs may never get a chance.
  - Shortest Remaining Time (SRT): Preemptive. May cause starvation.

- Scheduling Algorithms for Interactive Processing
  - Interactive processing: With active users interacting with the system.
  - Criteria: (1) Response time, (2) Predictability.
  - Round Robin (RR): FIFO with a fixed time slice (quantum). Guarantees response time.
    - Shorter time interval → More responsive but longer time spent by OS in context switching → Increasing average turnaround time
    - Behaves identically to FCFS if the job lengths are shorter than the time quantum.
  - Priority Scheduling: May cause starvation.
  - Multi-Level Feedback Queue (MLFQ): New job has the highest priority. If a job fully utilises its time slice, its priority reduces. If a job gives up or blocks before finishing its time slice, its priority retains. Jobs with same priority runs in RR.
    - Favours I/O intensive processes
  - Lottery Scheduling:

- Real-time Processing
  - Real-time processing: Have a strict deadline to meet.

2.3. Inter-Process Communication

- IPC Mechanism – Shared Memory
  - General idea: P1 creates a shared memory region and P2 attaches it to its own.
  - Advantages: (1) Efficient, (2) Ease of use.
  - Disadvantages: (1) Limited to a single machine, (2) Requires synchronisation to avoid data races.

- IPC Mechanism – Message Passing
  - General idea: P1 sends a message and P2 receives the message.
  - The message must be stored in kernel memory space.
  - Direct/Indirect Communication (via a mailbox)
  - Synchronisation behaviours:
    (1) Blocking: Sender is blocked until the message is received; Receiver is blocked until a message has arrived. Non-blocking send message is buffered in system.
    (2) Non-Blocking: Sender resumes immediately; If no message has arrived, receiver proceeds empty-handed but does not block.
  - Advantages: (1) Applicable beyond a single machine, (2) Portable, (3) Easier synchronisation (send/receive).
  - Disadvantages: (1) Inefficient, (2) Harder to use.

- UNIX Pipes: 1 end for reading, 1 end for writing. Pipes may be unidirectional or bidirectional, depending on UNIX version.

- UNIX Signal: A form of IPC sent to a process/thread. The recipient must handle the signal by a default set of handlers or user-supplied handlers.

2.4. Alternative to Process - Thread

- General Idea: Adds more threads of control to the same process so that multiple parts of the process are executing at the same time. Threads in the same process share memory context (text, data, heap) and OS context (PID, files, etc.). Each thread has unique ID, registers and stack. Only hardware context is switched in thread switch.

- Advantages
  - Economic: Much less resources compared with multiple processes.
  - Resource Sharing: Threads share most of the resources of a process.
  - Responsiveness: Much more responsive.
  - Scalability: Take advantage of multiple cores/CPUs.

- Disadvantages
  - Synchronisation around shared memory gets even worse (all except stack region).
  - System call concurrency
  - Process behaviour

- Thread models
  - User threads: Implemented as user library.
    Advantages:
    (1) Can have multi-threaded programme on any OS.
    (2) Thread operations are just library calls.
    (3) Generally more configurable and flexible.
    Disadvantages: OS is not aware of threads. One thread blocked leads to all threads blocked. Cannot exploit multiple CPUs.
  - Kernel threads: Implemented in OS.
    Advantages: Kernel can schedule on thread levels.
    Disadvantages:
    (1) Thread operations are system calls (slower and require more resources).
    (2) Generally less flexible.

2.5. Synchronisation

- Race Condition: Incorrect execution due to the unsynchronised access to a shared modifiable resource.
  - Solution: Designates code segment for critical section that allows at most one process.
  - Properties of critical section:
    (1) Mutual exclusion
    (2) Progress: If no process is in the critical section, one waiting process is granted access.
    (3) Bounded wait: There exists an upper bound for waiting time.
    (4) Independence: Process not executing in critical section does not block other process.
  - Symptoms of incorrect synchronisation:
    (1) Incorrect behaviour
    (2) Deadlock: All processes blocked.
    (3) Livelock: Processes keep changing states to avoid deadlock but make no other progress.
    (4) Starvation

- Critical Section Implementations
  - High-Level Programming Language: Uses only normal programming constructs.
  - Peterson's Algorithms:

```
Want[0] = 1;
Turn = 1;
while (Want[1] &&
       Turn == 1);
Critical Section
Want[0] = 0;
```
Process **P0**

```
Want[1] = 1;
Turn = 0;
while (Want[0] &&
       Turn == 0);
Critical Section
Want[1] = 0;
```
Process **P1**

  - Assembly-Level: Mechanisms provided by the hardware through special instructions.
  - High-Level Abstractions: Provides abstracted mechanisms that provide additional useful features.
  - Semaphore:

Wait(S)
- If S <= 0, **blocks** (go to sleep)
- **Decrement** S
- Also known as `P()` or `Down()`

Signal(S)
- **Increments** S
- Wakes up one sleeping process if any
- This operation **never** blocks
- Also known as `V()` or `Up()`

- $S_{current} = S_{initial} + \#signal(S) - \#wait(S)$
- Mutex: $S = 0$ or 1

```
int  atomic_increment( int* t )
{
   do {
   int temp = *t;
   } while (!_sync_bool_compare_and_swap(t, temp, temp+1));
   return temp+1;
}
```

2. [Critical Section] Can disabling interrupts avoid race conditions? If yes, would disabling interrupts be a good way of avoiding race conditions? Explain.

ANS:

Yes, disabling interrupts and enabling them back is equivalent to acquiring a universal lock and releasing it, respectively. Without interrupts, there will be no quantum-based process switching (because even timer interrupts cannot happen); hence only one process is running. However:

- This will not work in a multi-core/multi-processor environment since another process may enter the critical section while running on a different core.
- User code may not have the privileges needed to disable timer interrupts.
- Disabling timer interrupts means that many scheduling algorithms will not work properly.
- Disabling non-timer interrupts means that high-priority interrupts that may not even share any data with the critical section may be missed.
- Many important wakeup signals are provided by interrupt service routines and these would be missed by the running process. A process can easily block on a semaphore and stay blocked indefinitely, because there is nobody to send a wakeup signal.
- If a program disables interrupts and hangs, the entire system will no longer work since it cannot switch tasks and perform anything else.

Therefore disabling interrupts is not a good choice for the purpose of locking.

```
1  /* Define a pipe-based lock */
2  struct pipelock {
3     int fd[2];
   };
11 /* Initialize lock */
12 void lock_init(struct pipelock *lock) {

     pipe(lock->fd);
     write(lock->fd[1], "a", 1);

   //The first write is meant to initialize the lock such
     that exactly one thread can acquire the lock.

   }
21 /* Function used to acquire lock */
22 void lock_acquire(struct pipelock *lock) {

     char c;
     read(lock->fd[0], &c, 1);

   //read will block if there is no byte in the pipe.

   //Closing the reading or writing end of the pipe in a
     thread will cause closing that end for all threads
     of the process (shared variable). Also, it will
     prevent all other threads to acquire or release the
     lock.
   }
31 /* Release lock */
32 void lock_release(struct pipelock * lock) {

     write(lock->fd[1], "a", 1);

   //Need to write/read exactly one byte to simulate
     increment/decrement by 1 of a semaphore. It might
     work with multiple bytes, but you need to take care
     how many bytes are read/written.

   }
```