Operating Systems

Types of operating systems:

1. Batch
    1. non pre emptive
    2. No process can start until previous completely finished
    3. E.g.: Fortran, IBSYS 709X
2. Multiprogrammed
    1. Max possible processes inside RAM
    2. Non preemptive (less idle time than batch)
3. Multitasking
    1. Preemptive / time sharing

Process States

preemptive -> temporarily interrupt existing process (arrow from running state back to ready state)
non-preemptive -> no interrupt

1. New (create new process, popped to active/ready state)
2. Ready (queue; process inside RAM)
    1. Process brought to ready queue by LTS (long term scheduler)
    2. Bring max processes in RAM (multiprogramming concept)
    3. Dispatch / schedule to running state
    4. Depends on processor: how many processes dispatched
3. Suspend ready ***extra state***
    1. If ready queue is full, send high priority process to suspend-ready state
    2. When ready queue has space available, bring process to ready queue
    3. Medium term scheduler used here again
4. Running
    1. CPU executes
    2. Process still in RAM but CPU executing
    3. All instructions completed => terminated state
    4. If meanwhile, a higher priority process comes in ready queue, send current running process back to ready queue
    5. First execute high priority process
    6. In case time quantum expires, send current process back to ready queue and take next process (short term scheduler **(aka dispatcher)** -> bring process from ready state to running state)
    7. In case of I/O request, go to wait / block state
5. Wait / block (queue)
    1. state in RAM
    2. after completion of I/O task, go to ready queue
    3. If filled,
6. Suspend wait ***(extra state)***
    1. Process sent to secondary memory (suspended)
    2. Complete I/O task
    3. After completed and availability in wait queue, go to wait else stay in suspend state for a while
    4. Medium term scheduler -> if RAM keeps on filling, swap out process from RAM to secondary memory
    5. Backing store: send to suspend-ready state if wait queue is filled for a long time
7. Terminated
    1. deallocate resources

Note: LTS optimises cpu and io processes

Assign file permissions:

chmod _ _ _ _ _ _ _ _ _
Sets of 3: user, group, other
read, write, execute (r, w, x)

chmod go + r // give read permission to group and other
chmod ugo = r // read permission to all
chmod u + r, g + r, o - x

Octal:
read: 4
write: 2
execute: 1

chmod ugo + rw => chmod 666

System call: programmatic way of requesting resources / functionaliities from kernel / shift to kernel mode

Code executed => programmable process
When process is in running mode, it enters RAM.
Kernel is inside RAM.
Process invokes system call, gets permission from kernel, and accesses a file

Types of system calls:

1. File manipulation
    - open(), read(), write(), close(), create file, etc.
2. Device manipulation
    - read, write, reposition, ioctl,fctl
3. Information related
    - getpid(), getppid(), get system time / data
4. Process control
    - load, execute, abort, fork, wait, signal, allocate, etc.
5. Communication
    - Inter-process communication
    - e.g.: pipe(), shmget() (shared memory value get)

FORK

Child process = clone of parent having different id
In case of thread, entire process is not cloned

fork() returns 0, +1, -1

0 => child process
+1 => parent process
-1 => child process not created

User Mode vs Kernel Mode

UM => mode bit = 1
KM => mode bit = 0

PROCESS	THREAD
involves system calls	system calls not involved (user level)
OS treats different processes differently	All user level threads treated as single task
Different processes have different copies of data, files, code	Threads share same copy of code and data (each thread has own stack and registers)
Context switching slow	Fast
Blocking a process will	Blocking a thread will block entire process
Independent	Interdependent

Here, context switching:

Run process 1 for a while, then 2, then 1
Intermediate resource saving (hence slower)

Hardware or I/ O request => block (temporarily)

User level thread	Kernel level thread
managed by user level library	managed by OS (system calls)
fast	slower
faster context switching	slower
one blocked (hardware or I/O request) => entire process blocked	one blocked => no effect on others

Context switching time: process > KLT > ULT

ULT to KLT (m:1 relation) (Linux uses 1:1)

Process Scheduling algorithms:

1. Pre-emptive
    1. Shortest remaining time first (SRTF)
        - SJF with preemption
    2. Longest remaining time first (LRTF)
    3. Round robin
        - criteria: time quantum
        - all questions have running queue (single gantt chart)
        - make different ready queue for RR
        - first check if new process awaits
        - If yes, first put that in ready queue then put incomplete process from running to ready queue
        - how many times did context switching happen? (save current state of running process and push to ready queue)
        - each line except first and last in Gantt chart represents a context switch
    4. Priority based
        - criteria: priority
        - Important question with burst time of I/O also given
        - ratio of CPU idleness or usage: idle or (total - idle) divided by total
2. Non-preemptive
    1. First come first serve (FCFS)
        - criteria: arrival time
    2. Shortest job first (SJF)
        - criteria: burst time
    3. Longest job first (LJF)
    4. Highest response ratio next (HRRN)
    5. Multilevel queue
        - **RR** Highest priority: system process
        - **SJF** Medium: interactive (e.g.: streaming / video etc)
        - **FCFS** Low: batch process (background process etc)
        - Advanced version: **Multilevel Feedback Queue**
            - https://www.youtube.com/watch?v=-94WGbrWvel&list=PLxCzCOWd7aiGz9donHRrE9I3Mwn6XdP8p&index=23
            - process sent from lower PQ to higher priority queue as feedback
            - used to remove starvation in multilevel queue
            - Starvation = indefinite postponement of a process as it requires certain resource to run, but that resource is never allocated to the process
    6. Priority based

CPU Scheduling

Arrival time: time at which process enters ready queue
Burst / execution time: time (**duration**) required for process to get executed on CPU
Completion time: time at which process completes execution
Turn around time: completion time - arrival time
Waiting time: turn around time - burst time
Response time: time at which process got CPU first time - arrival time

Thread Scheduling Approaches

1. Load Sharing
    - load distributed equally across processors, assuring that no processor is idle while work is available to do
    - no centralized scheduler required

Process Synchronization

Process types:

1. Cooperative
    - execution of one affects another
    - share variables
    - share memory / buffer
    - share resources (CPU, I/O, etc)
2. Independent

Process synchronization is necessary in case of parallel cooperative processes, as race condition occurs (both processes "race" to give different value of shared resource and both may be wrong).

Producer-Consumer Problem

Both processes arrive at same time, and same resources (cooperative)

```
void Consumer() { // consumes items and finally pops from buffer and processes
    int itemc;
    while (true) {
        while (count == 0); // buffer empty
        itemc = Buffer(Out);
      Out = (Out + 1) mod n;
      Count -= 1;
      Process_item(itemc);
        // I1 load Rc, Count
        // I2 dec Rc
        // I3 store Count, Rc
   }
}

int Count = 0;  // global variable shared by both
// buffer common to both, shared

void Producer () { // produces items and puts to buffer
    int itemp;
    while (true) {
        Produce_item(itemp);
        while (count == n);  // buffer full
        Buffer[in] = itemp;
        in = (in + 1) mod n;
        Count += 1;
        // I1 load Rp, Count
        // I2 inc Rp
        // I3 store Count, Rp
    }
}
```

(Best case) -> produce 1, consume 1

Producer arrives, executes I1, I2
Before executing I3, consumer arrives, so producer preempts
Consumer execution begins, executes I1, I2
Before executing I3, producer resumes and executes I3
After producer terminates, consumer I3 executes

Critical Section

Part of program where shared resources are accessed by various processes

NCS -> EntryS -> CS -> ExitS in each process

Conditions / Rules to achieve Synchronization Mechanism

1. Mutual exclusion (**primary** rule) - ***MUTEX***
    - P1 is inside => P2 outside and vice versa
    - No two processes can be inside critical section at same time
2. Progress (**primary** rule)
    - CS not being executed by any process even though process wants to use => NO progress
3. Bounded wait (**secondary** rule)
    - P1 executes multiple times, but P2 never executes -> not allowed (type of starvation)
    - Infinite, 1 or 2 also not allowed
    - Above two subpoints are unbounded wait
4. No assumption related to hardware / speed (**secondary** rule)
    - Given solution works on 32-bit but not 64-bit
    - Speed reqd min 1GHz
    - Above two subpoints are assumptions which are not good

Semaphores

Method / tool to prevent race condition

Definition: Integer variable used in mutually exclusive manner by various concurrent cooperative processes to achieve synchronization

1. Counting semaphore (-inf to +inf)
2. Binary semaphore (0 or 1)

Operations in entry and exit code:

P(), Down, wait -> entry code (any of these can be used)

V(), up, signal, Post, Release -> exit code

Counting semaphore

```
Down(Semaphore S)
{
   S.value -= 1;
   if (S.value < 0) {        // if value is non-negative, let process enter CS
      put process(PCB) in suspended list  // process blocked
        sleep();
     } else return;
}

up(Semaphore S)
{
    S.value += 1;
    if (S.value <= 0) {     // if non-positive,
        select a process from suspended list &&
           wake up();    // push to ready queue, since direct running queue
    }
}
```

```
Note that if S.value >= 0, no process in suspended list
If S.value = 0, feel free to make it = 1, as that is what will happen after executing up() anyway.

S.value => that many number of processes can enter CS successfully
Blocked => unsuccessful
```

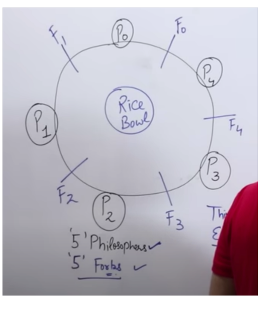
Binary semaphore

```
Down(Semaphore S)
{
    if (S.value == 1)
        S.value = 0;
    else {
        Block this process
        and place in suspend list,
        sleep();
    }
}

up(Semaphore s)
{
    if (suspend list is empty)
        S.value = 1;
    else {
        Select a process from
        suspend list,
        wake up();
    }
}
```

Producer-Consumer using Binary Semaphore

```
void Producer()
{
    down(empty);
    down(s);

    Buffer[in] = itemp;
    in = (in + 1) mod n;

    up(s);
    up(full);
}

void Consumer()
{
    down(full);
    down(s);

    itemC = Buffer[out];
    out = (out + 1) mod n;

    up(s);
    up(empty);

}
```

Dining Philosopher's Problem (Semaphore + Deadlock)




```
void Philosopher() {

    // S[N] is array of binary semaphores
    // S[i] == 1 => fork can be taken; else fork is being used

    while(true) {

        thinking();
        wait(take_fork(S[i])); //  take left fork (S[i] = 0)

        // if all processes (P0 to P4) preempt after this step, deadlock will occur

        wait(take_fork(S[(i + 1) % N]));   // take right fork (S[(i + 1) % N] = 1)
        eat();
        signal(put_fork(S[i]));   // put left fork (S[i] = 1)
        signal(put_fork(S[(i + 1) % N]));  // put right fork (S[(i + 1) % N] = 0)

        // solution: if (i == N - 1) take right fork first then take left fork => prevents deadlock

    }

}
```

Deadlock

If two or more processes are waiting for occurrence of an event, which never happens

Conditions necessary for deadlock to occur:

1. Mutual exclusion (interleaving not allowed)
2. No preemption
3. Hold and wait (holding one resource, waiting for another resource)
4. Circular wait (cycle) => **ALWAYS DEADLOCK** for all single instance resources

Resource Allocation Graph

Process -> vertex with circle
Resource -> vertex with rectangle -> two types: multiple and single instance

Assign edge: moving towards process
Request edge: moving towards resource

To check if deadlock is present:
Make a table with columns (process_number, allocate, request)
Allocate and request both have subcolumns with resource number i in ith column
Check availability
If process requests nothing, it gets terminated and allocated resource is freed,
hence **availability changes**

Note: Finite waiting == **starvation**; infinite waiting = **deadlock**
Note: deadlock occurs once in 5 / 10 years

Deadlock Handling Methods

1. Deadlock ignorance (ostrich method)
    - **widely used** by Linux, Windows
    - restart / power off
    - name, because in case of sandstorm ostrich puts his head in the sand, under the ground, to ignore sandstorm
    - performance is affected by adding an algorithm to handle deadlock explicitly, hence this is used

2. Deadlock prevention
    - negate all conditions
    - enable preemption
    - disable mutex
    - before process gets started, provide all resources required (practically impossible) => no hold & wait

3. Deadlock avoidance (Banker's algorithm)
    - Process informs system beforehand: max resources required
    - 5 columns (process_number, allocation, max_need, available, remaining_need = max_need - allocation
    - Each column other than process_number has n sub-columns for resource r(i) till n
    - First compute remaining_need column
    - According to availability, choose first process having remaining_need <= availability to terminate
    - Availability += max_need of executed process (if subtracted remaining_need)
    - Availability += allocated (if not subtracted remaining_need)
    - Sequence of process execution need not be unique
    - If all processes get executed, it is called a **safe sequence**
    - **Max_need** cannot be static, it is always dynamic

4. Deadlock detection and recovery
    - very complex
    - Detection:
        - Kill process(es) till deadlock exists
        - Hampers system performance
        - Resource preemption

Memory Management

CPU is connected to:

- Registers
- Cache
- RAM
    - Connected to secondary memory

- CPU has to bring processes from secondary memory to RAM (swap), to directly interact and execute
- Higher degree of multiprogramming => higher usage of CPU

Techniques

A. Contiguous

1. Fixed (static) partition
    - Used in 1960s in mainframes
    - Size of each partition need not be the same
    - **Number of partitions is fixed**
    - Spanning is not allowed => one process cannot span over multiple partitions
    - Process can be put in any available partition
    - *Internal fragmentation*: Size required by process < partition size
    - *Limit in process size*: Process requiring size > max partition size cannot be accommodated
    - *Limited degree of multiprogramming*: Max. number of processes = number of partitions
    - *External fragmentation*:
        - New process requirement is available in RAM but not in one partition (spanned over multiple partitions)
        - Whenever there is internal fragmentation, external always exists
2. Variable (dynamic) partition
    - Process demands size while entering
    - Dynamically allocated in RAM
    - Internal fragmentation NOT possible
    - Degree of multiprogramming: no limitation
    - Process size: no limitation
    - *Hole*:
        - Middle process completed / exited => hole created;
        - cannot be occupied since contiguous allocation
        - Holes are created of fixed size (however allocated dynamically)
    - *External fragmentation* occurs in case of holes
        - Process requirement space available collectively (but not contiguously) so process has to wait
        - **Compaction** can be used; but is undesirable
            - Collect all holes together
            - Running processes have to stop
            - Processes have to be copied to different location, time-heavy
    - *Allocation / Deallocation* is complex
        - Assigning a process to a hole
        - Managed using bitmap / linked list

Algorithms to allocate processes:

1. First-fit
    - Allocate first hole that is big enough
    - Each search starts from 0 index
    - Simple, fast
2. Next-fit
    - Same as first-fit but start search always from last allocated hole
    - Each search begins from pointer, instead of 0
    - Faster than first-fit
3. Best-fit
    - Search entire list, find hole leading to minimum internal fragmentation
    - Creates such tiny holes that any process in future may not be accommodated
    - Slow
4. Worst-fit
    - Search entire list, find hole leading to maximum internal fragmentation
    - Hole size is large enough to accommodate future processes
    - Slow

B. Non-contiguous

- Partition of RAM is called frame
- Partition of process is called page
- Process is partitioned while in secondary memory itself
- Page size == frame size **ALWAYS**

1. Paging
2. Multilevel paging
3. Inverted paging
4. Segmentation
5. Segmented paging