Lecture 2. Notes Summarised.

Define a Process:

* A program in execution
* Executable file containing list of instructions is passive (unless instructions are executed)

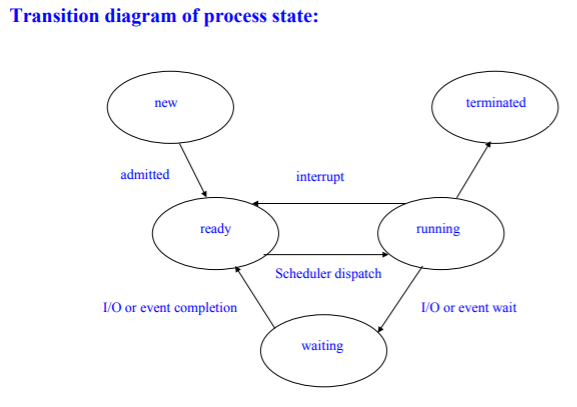
Requirements of a process:

1. Requires a counter, it specifies the next instruction to execute
2. Needs resources (CPU time, memory, files & I/O devices) to accomplish its task

What does a Process Include:

1. Text Section – the program code
2. Program Counter – pointer to next next instruction to execute
3. Contents of process registers
4. Stack: contains temporary data (temp variables, return addresses etc..)
5. Data Section: global Variables
6. Heap: memory dynamically allocated during run time.

Responsibilities of the Process Management of an OS:

* Creation and deletion of user and system processes
* Suspension and resumption of processes
* Provision of mechanisms for process synchronization (coordinate the process that uses shared data)
* Provision of mechanisms for process communications
* Provision of mechanisms for deadlock handling

Two types of processes:

* User Level
* Kernel Level

Different types of states a Process can be in:

* New – process is being created
* Running – instructions are being executed
* Waiting – the process is waiting for some event to occur
* Ready – the process is waiting to be assigned to a processor
* Terminated – the process has finished execution

Two Data Structures that represent a process:

1. Process Control Block (PCB)
2. Task Control Block

Define Process Control Block:

Data structure in the operating system kernel containing the information needed to manage the scheduling of a particular **process**.

Characteristics of a PCB:

1. Process Number (pid)
2. Process state: new, ready, running, waiting etc
3. Program counter: shows address of next instruction to be executed
4. CPU registers: vary in number and type (depend on computer architecture)
5. CPU scheduling information: process priority, pointers to schedule queues, etc
6. Memory management information: base and limit registers
7. Accounting Information: amount of CPU and time used
8. I/O status information: list of I/O devices allocated to this process, list of open files etc..

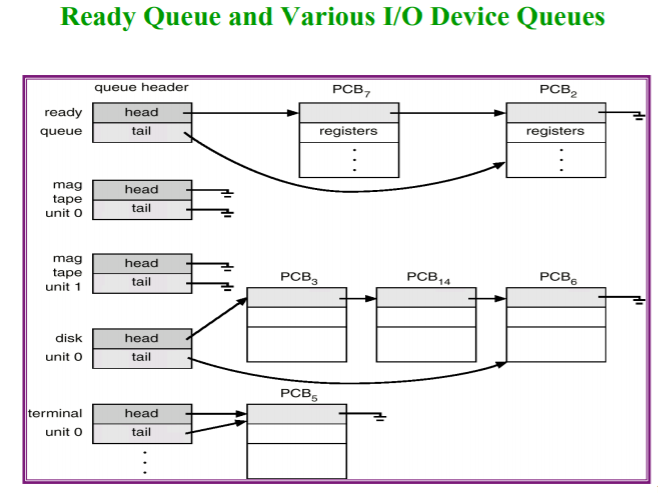
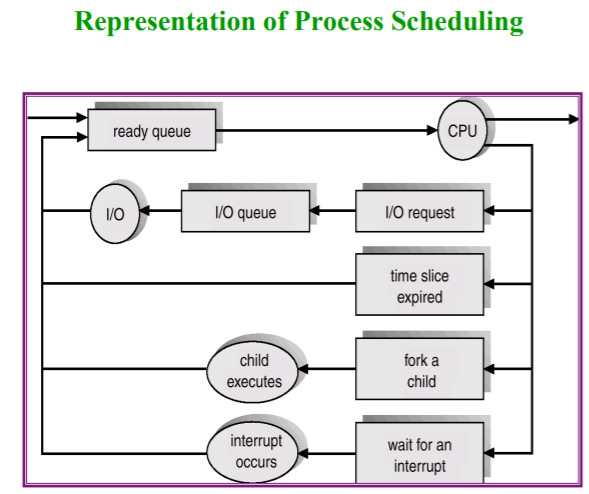
Hierarchy of a Process:

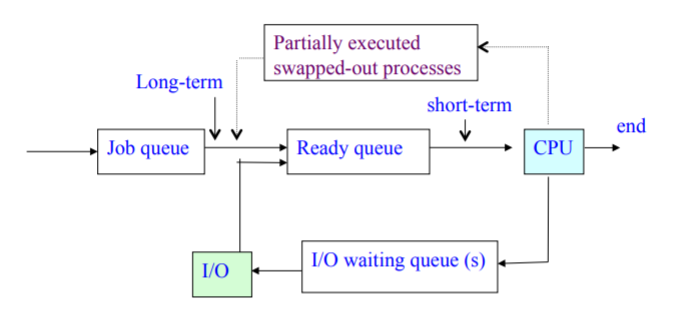
* **Parent:** parent of a process is a process that creates it
* **Children:** Process that is created by parent
* **Siblings:** children with same parent

Purpose of the CPU scheduler in different systems:

1. Multi-Programming Objective: maxmise CPU utilization
2. Time Sharing Objective: aims to switch processes frequently so that users can interact with their running programs

Several Queues (implemented as linked lists) that are stores in OS:

* Job Queue: All processes entering the system are put in a job queue
* Ready Queue: A set of all processes residing in main memory, ready and waiting to execute
* Device Queue: A set of processes waiting for an I/O device (each device has its own queue)



Function of a Long-Term Scheduler:

* Selects which processes should be brought into the ready queue.
* Controls the degree of multiprogramming (number of processes in memory)
* Invoked very frequently

Function of a short-term scheduler:

* Selects or dispatches a process to be executed next and next and allocated CPU
* Invoked very frequently; thus scheduler must be fast

The two description of processes:

1. I/O bound process:

* Spends more time doing I/O than computation
* Has many short CPU bursts

1. CPU bound process:

* Spends more time doing computations
* Very long CPU bursts

Define Context Switch:

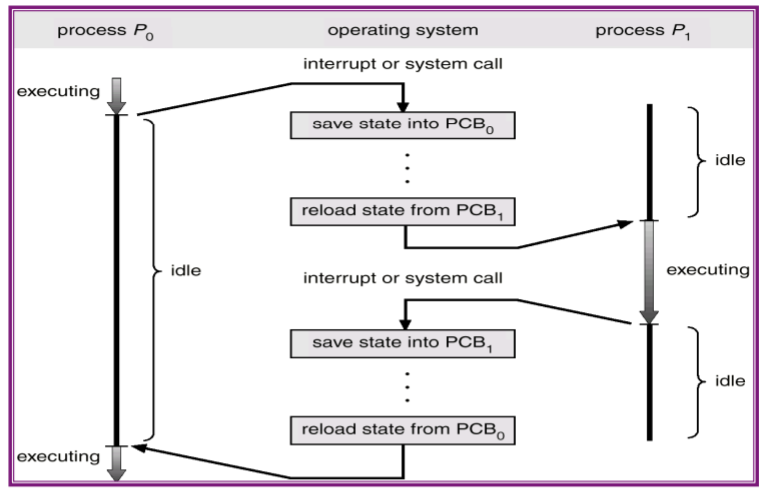
Process of storing the state of a process or of a thread, so that it can be restored and execution resumed from the same point later. This allows multiple processes to share a single CPU, and is an essential feature of a multitasking operating system.

Disadvantage of Context Switches:

1. Overhead
2. System does no useful work while switching
3. Performance bottleneck
4. Dependent on hardware support

When do we switch CPU ?

* Job voluntarily waits (system calls)
* Interrupt: higher priority event/job needs attention
* Interrupt: timer 🡨what does that mean ? program counter perhaps?



**CPU switch from process to process**

Types of Process Operation:

1. Process Creation
2. Process Termination

Describe Process Creation:

Parent process creates a child processes, which in turn can create other processes, forming a tree of processes.

How is each Process identified:

* By process ID (an integer value)
* The init process has PID=1; it is the root parent process for all other user processes.
* A process requires resources (CPU, memory, files, I/O)

What happens when a process creates a Sub-process:

1. Resources:

* Parent & child may share all resources
* Child processes may share subset of parents resources
* Parent & child process may share no resources

1. Execution:

* Parent and child process may execute concurrently (linux)
* Parent may wait until child process terminates (parent calls wait() to move itself out of the way)

1. Address Space:

* Child process address space is duplicate of parent
* Child has a program loaded into it

Describe Process Termination:

* A process terminates when it executes the last statement and/or calls exit()
* OS will deallocate the child’s resources
* The child cab then return its status value to parent that calls pid = wait(&status)

What happens if the parent does not call the wait()

* OS will not remove the child from process table
* Child process becomes a ‘zombie’
* When parent terminates, child becomes an orphan, and init will automatically be its parent

(init periodically calls wait() to remove orphans from process table)

Reasons as to why Parent may terminate execution of child process (abort):

* Child has exceeded allocated resources
* Task assigned to child is no longer required
* Parent is exiting (Some OS don’t allow child to run if parent terminates)

Two Types of Concurrent processes executing in the OS:

1. Independent Processes: Cannot affect or be affected by the execution of another process
2. Cooperating Process: Can affect or be affected by the execution of another process

Advantages of Process Cooperation:

* Information Sharing: several users may need same piece of information
* Computation Speed: break a task into subtasks and execute in parallel
* Modularity: Divide the system function into separate processes
* Convenience: A user may want to do editing, printing in parallel

How do Cooperating Processes Communicate?

* Shared Memory
* Message Passing

Shared memory Vs. Message Passing:

|  |  |
| --- | --- |
| Shared Memory | Message Passing |
| * Process uses a system call to create a shared memory region * Other process that wants to communicate via memory region must attach to it, its address space * Once established, all accesses to shared memory are as if assessing normal memory area (no system call) * Suffers from cache coherency issues * Note: all accesses must be synchronized | * Uses a system call to exchange message between processes * More time consuming * OS handles synchronization between processes * Good for distributed system with small amount of data exchanged |

The types of buffers that can be filled and emptied by processes

1. Unbounded Buffer: places no practical limit on size of buffer
2. Bounded Buffer: assumes there is a fixed buffer size

Note: These buffers can be in the form of shared memory, or provided by OS vis message passing

Two operations that Message passing facilitates:

* Send (message) – message size fixed or variable
* Receive (message)

What happens if two operations would like to communicate?

1. Establish a communication link between them
2. Exchange messages via send/ receive

Methods for logically **implementing** a link and send/receive operations:

* Direct/indirect communication
* Synchronous/ Asynchronous communication
* Automatic/ explicit buffering

How does Direct communication work?

* Each process must explicitly name the recipient or sender of the communication
* Primitives Used:

1. Send( P, message) 🡪 send a message to process P
2. Receive (Q, message) 🡪 receive a message from process q

Properties of Communication link for direct communication:

1. Links are established automatically
2. A link is associated with exactly one pair of communicating processes
3. Between each pair there exists one link
4. The link may be uni-directional, but is usually bi-directional

How does Indirect Communication work?

* Each mailbox has a unique ID
* Processes can communicate only if they Share a mailbox
* Primitives used:

1. Send (A, message) 🡪 send a message to mailbox A
2. Receive(A, message) 🡪 receive a message from mailbox A

Properties of communication link for this scheme:

1. Link only established if processes share a common mailbox
2. A link may be associated with many processes
3. Each pair of process may share several communication links
4. Link may be uni-direction or Bi-directional

Operations in Indirect Communication:

1. Create a new mailbox
2. Send and receive messages through the mailbox
3. Destroy a mailbox

Mailbox Sharing:

Consider processes P1, P2, and P3 share a mailbox A, and process P1 sends a message to mailbox A while processes P2 and P3 execute a receive from A,

Which process will receive the message sent by P1?

Three Solutions include:

1. Allow a link to be associated with at most two processes
2. Allow only one process at a time to execute a receive operation
3. Allow the system to select the receiver arbitrarily. Send is notified of who receiver is

Types of Message Passing:

* Blocking (Synchronous)
* Nonblocking(Asynchronous)

1. Blocking Send: Sender is blocked until message is received
2. Non-blocking Send: sender resumes operation after sending message
3. Blocking receive: the receiver blocks until message is available
4. Non-blocking receive: receiver retrieves either a valid message or a NULL

Buffering (Slide 31)

Exception Conditions and Recovery Steps that must take place:

Cause: Error may happen during process communication. Failure occurs in a centralized/distributed machine, recovery must take place after.

1. Problem: Process terminates e.g. Process P is waiting for a message from a terminated process Q

Solution: OS terminates P or notify that P that Q has terminated

1. Problem: Lost Messages e.g. message from P to Q become lost due to hardware error (most common lost detection is timeout)

Solution: OS detects it and resends message

Sender detects it and resends

OS detects it and lets sender know

1. Problem: Scrambled message, message is received but in error (detected by parity checking)

Solution: resend it

Define Threads:

Small sequence of programmed instructions that tell the computer what it has to do to perform that command. It refers to the highest-level code your processor can execute. Threads are managed by a scheduler. Unit of execution within a process.

States of Threads:

1. Ready
2. Blocked
3. Running
4. Terminated

Characteristics of a Thread:

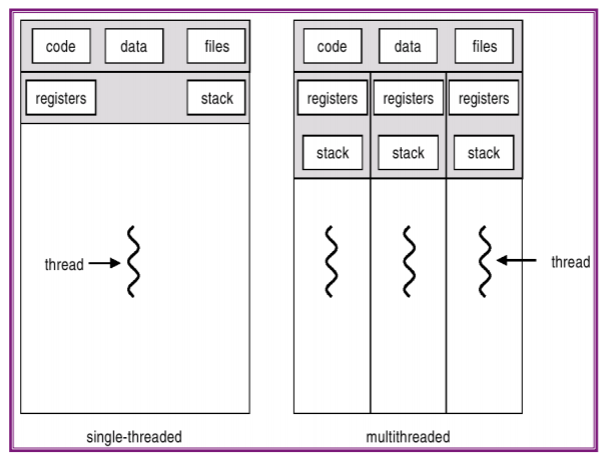
1. Basic unit of CPU utilization
2. Share all process memory and other resources
3. Threads share CPU (one thread at a time running)
4. Thread within a process executes sequentially
5. Each thread has its own Program Counter & Stack
6. Threads within a process are generally invisible from outside process
7. Threads within a process are scheduled and executed independently in the same way as different single-threaded processes

Difference of Threads:

* Not independent of one another
* All threads can access every address in the task
* No protection between threads (within a process)

Benefits of a thread:

1. **Responsiveness:** when one thread in the program is blocked and waiting, a second thread in the same task can run
2. **Resource Sharing:** Threads share memory and resources within a process
   1. Applications that require sharing a common buffer (e.g. producer-consumer) befit from thread utilization
3. **Economy:** It is more economical to create and context switch threads than processes
4. **Utilization of multiprocessor Architecture:** Each thread may run in parallel on a different processor

Single Vs. Multithreaded Process.

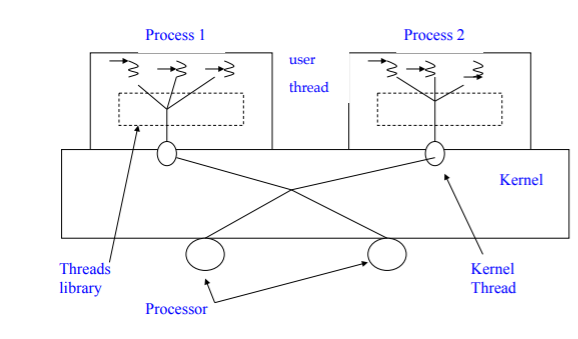
Types of Threads:

1. Kernel Supported Threads
2. User-level threads

|  |  |  |
| --- | --- | --- |
|  | **User Level Threads** | **Kernel-Supported Threads** |
| **Differences** | * implemented by users * kernel is not aware of the existence of these threads. * Thread management is done by user-level threads library * handles them as if they were single-threaded processes. * small and much faster than kernel level threads. * Represented by a program counter, stack, registers and a small process control block * no kernel involvement in synchronization for user-level threads. | * Kernel-level threads are handled by the operating system directly * thread management is done by the kernel * Only kind of thread kernel knows about * Don’t have their own scheduler * Expensive |
| **Advantages** | * 1. easier and faster to create than kernel-level threads.   2. They can also be more easily managed.   3. Can be run on any operating system.   4. There are no kernel mode privileges required for thread switching in user-level threads. | * 1. Multiple threads of the same process can be scheduled on different processors in kernel-level threads.   2. The kernel routines can also be multithreaded.   3. If a kernel-level thread is blocked, another thread of the same process can be scheduled by the kernel. |
| **Disadvantage** | * 1. Multithreaded applications in user-level threads cannot use multiprocessing to their advantage.   2. The entire process is blocked if one user-level thread performs blocking operation.   3. Kernel considers a set of user-level threads as a single thread   4. If any user level thread is blocked then other threads in set cannot run | * 1. A mode switch to kernel mode is required to transfer control from one thread to another in a process.   2. Kernel-level threads are slower to create as well as manage as compared to user-level threads. |

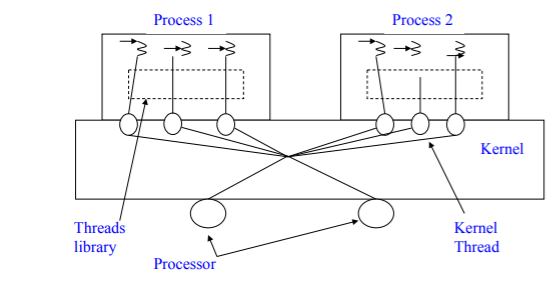
User to Kernel Threads Mapping:

* Many-to-one model
* One-to-One model
* Many-to-Many Model

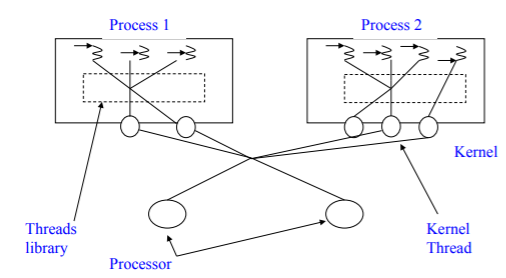
Many-to-One Model:

* User level threads are mapped to one Kernel Thread
* Multiple Threads within a process cannot run in parallel on multiprocessors
* Used by Solaris 2 and thread libraries for systems with no kernel threads.

One-to-One Model:

* Each user thread is mapped onto a kernel thread
* This allows concurrency; allowing more threads to run when a thread is blocked
* Allows multiple threads running in parallel on multiprocessors
* Example: Windows, Linux

Many-to-Many Model:

* Allows M user-level threads to be mapped to N kernel threads
* Allows the OS to create sufficient number of kernel threads
* Many user threads can be created as necessary and their corresponding kernel threads can run in parallel on multiprocessors

What is Thread Cancellation?

* When multiple threads are running in a process, thread cancellation permits one thread to cancel another thread in that process.
* The target thread (the one being cancelled) can keep cancellation requests pending and can perform application-specific cleanup when it acts upon the cancellation notice.

Two different Scenarios of Thread cancellation:

1. Asynchronous cancellation: One thread immediately terminates target thread
2. Deferred cancellation: target thread periodically checks if it should terminate

Problems with Cancellation:

1. If resources have been allocated to a cancelled thread, the Operating system reclaims those system resources (not all of them, asynchronous)
2. A thread was cancelled while in the middle of updating data shared with other threads

What is Signal Handling?

* A signal is used in Unix to notify a process that a particular event has occurred

Types of Signals:

|  |  |
| --- | --- |
| Synchronous Signals | Asynchronous Signlas |
| * Delivered to same process that performs the operation causing the signal * Operation can be divide by 0, illegal memory access | * Generated by an event external to running process * <control><c>, timer expired |

Pattern that signals follow:

1. Signal generation: occurrence of a particular event
2. Delivering signal to a process
3. Signal handling

Two Possible Signal Handlers:

1. Default Signal Handler: every signal has a default handler which is run by the kernel
2. User-defined signal handler: override the default

When a process has more than one thread where should signal be delivered ?

Options:

1. Deliver the signal to the thread to which the signal applies -> for synchronous signals
2. Deliver the signal to every thread in the process -> for some asynchronous signals such as <control><c>
3. Deliver signal to certain threads in the process -> options in some versions of UNIX
4. Assign a specific thread to receive all signals for the process -> implemented in Solaris 2

Disadvantages of not using Thread pools:

* Time consuming for creating & destroying a thread for each request
* No limit (bound) on the number of threads created
* May exhaust system resources ^

What are Thread pools:

Number of threads created at a process start-up that are placed in a pool waiting for work.

Benefits of Thread Pools:

* Faster to service a request
* Limits number of threads that may exist at any one point

How are the number of threads in a pool set?

1. Heuristically: Based on system resources, such as CPU, memory or based on number of requests
2. Dynamically: Adjusted on usage patterns