Distributed Process Scheduling

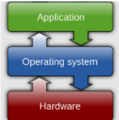
Topic 4

Hartmut Kaiser

https://teaching.hkaiser.org/fall2025/csc7103/

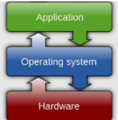
Scheduling Basics

- Before execution/running, processes must be scheduled for CPUs and allocated with resources
- Objectives of scheduling
  - Enhance overall system performance metrics such as process completion (turnaround) time and processor (resource) utilization
  - Achieve location and performance transparency
- Several challenges:
  - Multiple processing nodes are geographically distributed
    - Dynamical behaviors of the system – underlying architecture
  - Local and global scheduling
  - Distributed processes execute on remote nodes and may migrate from node to node:
    - Remote execution, process/task migration, data migration
  - Non-negligible communication overhead

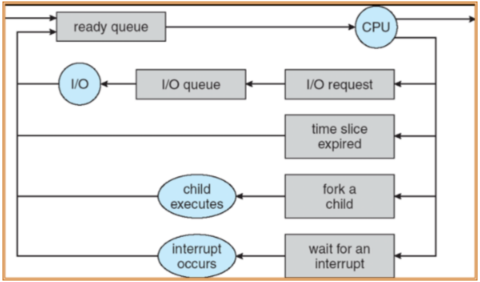


Process/CPU Scheduling

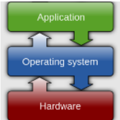
- Scheduler selects from among the processes in ready queue, and allocates the available processor/CPU to one of them
- Scheduling decision may take place
  - When a process switches state (e.g., running to waiting)
  - When a new process enters the system
  - Whenever a scheduling condition is met
  - When a process terminates
- Preemptive vs. non-preemptive
  - Scheduling can be non-preemptive (static) – once CPU is allocated to a process, the process keeps CPU until its execution finishes
  - Scheduling can be preemptive – process is interrupted (suspended) for other process to be allocated to CPU
- Dispatch latency – time the dispatcher takes to stop one process and start another running



Representation of Process Scheduling

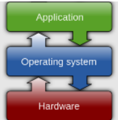


Useful formulation of scheduling: How is the scheduler to decide which of several tasks to take off a queue?



Common Scheduling Algorithms

- Several scheduling algorithms exist for single processor computer system:
  - First-Come, First Served (FCFS) scheduling
  - Shortest-Job-First (SJF) scheduling
  - Shortest-Remaining-Time-First (SRTF) scheduling (preemptive)
  - Priority scheduling (preemptive/non-preemptive)
  - Round Robin (RR) scheduling (preemptive)
  - Multilevel Queue scheduling
  - Multilevel Feedback Queue scheduling

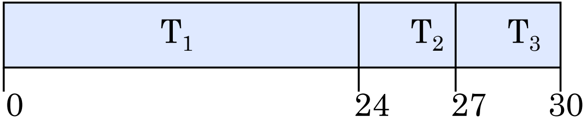


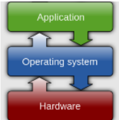
First-Come, First-Served Scheduling (FCFS)

- Also: “First In First Out” (FIFO)
- Example:

Process	Burst Time
T1	24
T2	3
T3	3

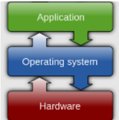
- Arrival Order: T1, T2, then T3 (all arrive at time 0)





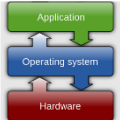
First-Come, First-Served Scheduling (FCFS)

- Response Times: T1 = 24, T2 = 27, T3 = 30
  - Average Response Time = (24+27+30) / 3 = 27
- Waiting times: T1 = 0, T2 = 24, T3 = 27
  - Average Wait Time = (0 + 24 + 27) / 3 = 17
- Convoy Effect: Short processes stuck behind long processes
  - If T2, T3 arrive any time < 24, they must wait



Slightly Different Arrival Order?

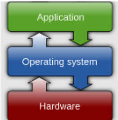
- T2 < T3 < T1
- Response Time: T1 = 30, T2 = 3, T3 = 6
  - Average Response Time = (30 + 3 + 6) / 3 = 13
  - Versus 27 with T1 < T2 < T3
- Waiting Time: T1 = 6, T2 = 0, T3 = 3
  - Average Waiting Time = (6+0+3) / 3 = 3
  - Versus 17 with T1 < T2 < T3



Optimization Criteria

- Speedup – Increasing throughput (the number of completed processes per time unit)
- Resource utilization – keeping processors as busy as possible
- Makespan or completion time – decreasing turnaround time (waiting + execution + event)
- Response time – reducing the time taken from request submission until the first response (important in time-sharing systems)
- Load sharing and balancing – key in distributed and multiprocessor systems

Optimize the max or min values or average measure or variance

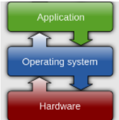


Speedup

- The purpose of partitioning processes is for speeding up executions
  - Speedup perhaps is the most important performance metric for distributed computing
- Speedup factor (S) is a function of the parallel/distributed algorithm, system architecture, and scheduling

$$S = S_i \times S_d = \left(\frac{RC}{RP} \times n\right)\left(\frac{1}{1+\rho}\right)$$

- **S_i** = ideal speedup, **S_d** = degradation due to actual implementation
- **n** = number of processors used
- **RC** = relative concurrency: how far from optimal is the usage of the processors
- **RP** = relative processing: how far from optimal is the speedup with a parallel algorithm compared to the ideal sequential algorithm
- ρ = Efficiency loss: loss of parallelism when implemented on a real machine (ratio of overheads and optimal processing time)
- S *reflects the real-world speedup for parallelizing a sequential program*



Speedup

- RC - relative concurrency: how far from optimal is the usage of the processors

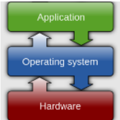
$$RC = \frac{\sum_{i=1}^{m} P_i}{CPT_{ideal} \times n}$$

  - P_i: processing time for step i
  - CPT_{ideal}: optimal concurrent processing time

- RP - relative processing: how far from optimal is the speedup with a parallel algorithm compared to the ideal sequential algorithm

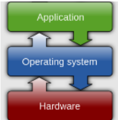
$$RP = \frac{\sum_{i=1}^{m} P_i}{SPT_{ideal}}$$

  - SPT_{ideal}: optimal sequential processing time



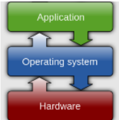
Multiple Processor Scheduling

- Scheduling becomes more complex when multiple CPUs/processors are available – load sharing/balancing is important
- Homogeneous (identical) processors within a multiprocessor system:
  - Asymmetric multiprocessing
    - One single processor (called master server like coordinator) does all scheduling including I/O processing
    - Other processors execute only user codes
  - Symmetric multiprocessing (SMP)
    - Each processor is self-scheduling.
    - Provide a separate queue for each processor
    - Use a common ready queue (naturally self-balancing)
- Distributed systems:
  - Processors may not be identical (heterogeneous system)
  - Communication overhead (heavy networking) is non-negligible
  - Clock and memory are not shared



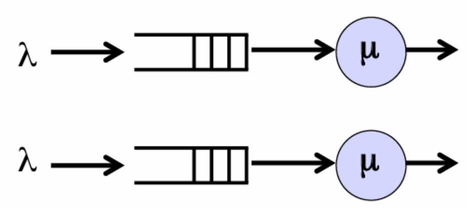
Scheduling Evaluation Methods

- Deterministic modeling
  - Takes a particular predetermined workload, and evaluate the performance metric for the workload
- Queuing models
  - Use statistical approaches
  - X / Y / c queue with an arrival process X, a service time distribution of Y, and c servers (Kendall's notation)
    - e.g., Poisson distribution for arrival time, exponential distribution for service time, etc.
  - Isolated workstation, processor pool, migration workstation
- Simulations – program a model of the computer system (a simulator)
- Implementation – code the algorithm, put in the real OS, and see how it works



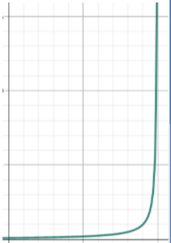
Isolated Workstation Model

- Maintains a separate queue for each workstation – static scheduling
- M/M/1 queue - no sharing of the workload is attempted
  - Arrival process distribution **M** / service time distribution **M** / one server
  - **M** represents a Markovian distribution



Average turnaround time:
$t = 1/(\mu - \lambda)$

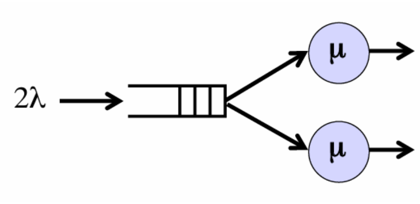
Turnaround time is the sum of service and queuing times due to waiting of processes



- **λ** and **μ** are the arrival and service rates of each processing node (i.e., 1/**λ**: mean arrival time, **1/μ**: mean service time)

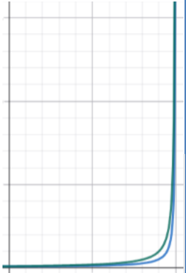
Processor Pool Model

- A process is dispatched to an available processor and remains there statically throughout the entirety of its execution
- M/M/2 queue – a waiting job can be serviced by either of two processors
  - Two servers model a system



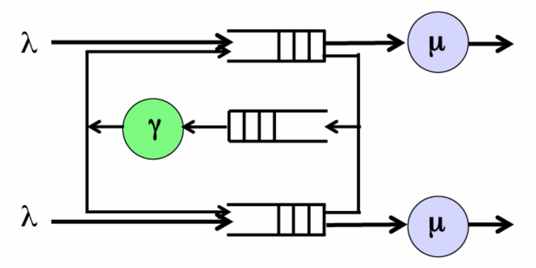
Average turnaround time:

$t = \mu / (\mu + \lambda)(\mu - \lambda)$

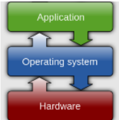


Migration Workstation Model

- In migration workstation model, processes are allowed to move from one workstation to the other
- Process migration incurs some communication overhead (modeled as an additional queue)
  - γ : process migration rate



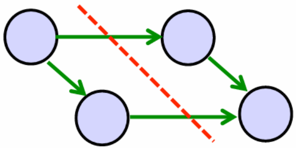
Average turnaround time lies between those of isolated workstation model and processor pool model (that is, better than M/M/1 but worse than M/M/2)



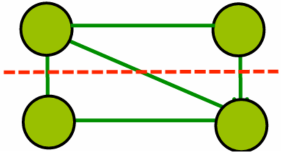
Distributed Scheduling

Process Interaction Models

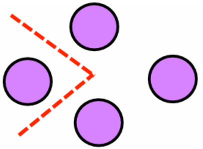
- Distributed scheduling algorithms represent sets of multiple processes governed by rules that regulate the interactions among processes
  - Partitioning of processes and how to map processes to processors
- Graph models describing process communication:
  - Precedence process model – satisfy precedence relationships
  - Communication process model – processes coexist and communicate asynchronously
  - Disjoint process model – process can run independently



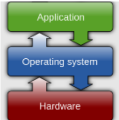
Precedence process model



Communication process model

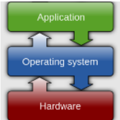


Disjoint process model



Distributed Scheduling Algorithms

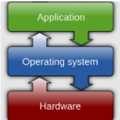
- Static (or off-line) scheduling
- Dynamic (or on-line) scheduling
  - Load sharing – static workload distribution
  - Load balancing – dynamic workload distribution
- Real-time scheduling



Static Scheduling

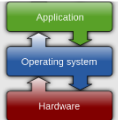
Static Process Scheduling

- Static process scheduling (deterministic scheduling theory) deals with a set of partially ordered tasks on a non-preemptive multiprocessor system
  - System consisting of **identical** processors
  - Goal of minimizing the **overall finishing time** – makespan
- The general problem is NP-complete
  - Scheduling to optimize makespan is NP-complete
  - Only approximate or heuristic methods to obtain near optimal solution
- In distributed systems, the problem becomes even more complicated because communication is not only **non-negligible** but is also a characteristic of the system. Moreover, processors are **not identical**



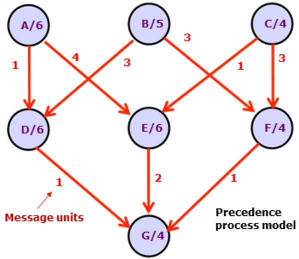
Static Process Scheduling (Cont.)

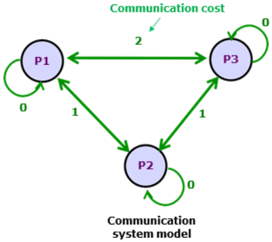
- Find good but easy-to-implement heuristic approaches for scheduling processes in distributed systems
  - Balance and overlap computation and communication
- In static scheduling, the mapping of processes is determined before the execution of the processes
  - Once a process is started, it stays at the processor until completion (static)
  - Need good knowledge about process behavior: process execution time, precedence relationships and communication patterns between processes
  - Scheduling decision is centralized and non-adaptive
  - Impacted by communication: Communication process model versus communication system model
- Static distributed process scheduling algorithms:
  - Stone's algorithm
  - Two heterogeneous processors, arbitrary communication process graph
  - n-processor generalization

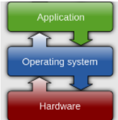


Using Precedence Process Model

- Static multiprocessor scheduling based on precedence process model
  - Minimizing overall makespan (overall finishing time)
- Using the DAG of the program, a heuristic algorithm tries to find a good mapping of the process model to the system model
  - Example: 7 tasks (A through G) with execution times and message units shown to be mapped on 3 processors (P1, P2 and P3) with non-negligible inter-processor communication costs/delays shown

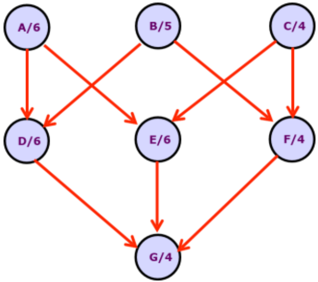


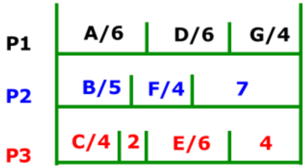




List Scheduling – Ignore Communication Overhead

- List Scheduling (LS): No processor remains idle if there are some tasks available that it could process
- Critical path – the longest execution path in the DAG, which is a lower bound of makespan
  - Critical path for the example is **ADG** or **AEG** of length = 6 + 6 + 4 = 16



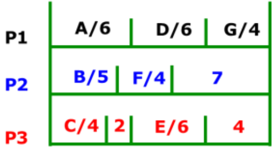


Makespan = 16

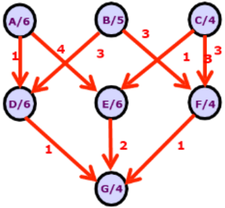
Optimal result though the algorithm is heuristic

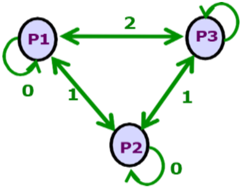
Extended List Scheduling – With Communication Overhead

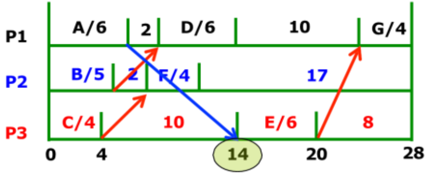
- Extended List Scheduling (ELS): First allocate tasks to processor by applying LS and then add the necessary communication delays
- Communication delays are computed by multiplying the unit communication cost and message units



LS makespan = 16 units





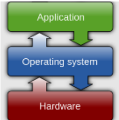


Cross lines represent waiting for communication

ELS makespan = 28 units

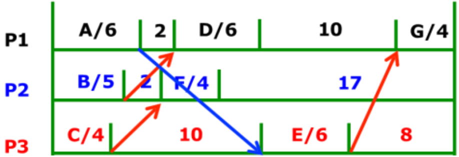
Far from optimal

Comm. Cost = 6

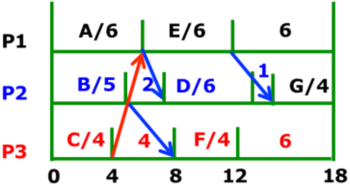


Earliest Task First Scheduling - Delaying Decision

- Earliest Task First (ETF) – the earliest schedulable task is (delayed) scheduled first
  - Communication costs are included in the calculation



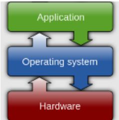
ELS makespan = 28 units



Delay scheduling task F because task E will become schedulable first

ETF makespan = 18 units

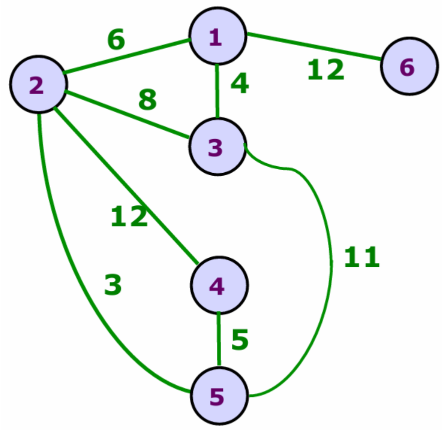
Comm. Cost = 5



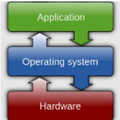
Using Communication Process Model

- The primary objective of distributed process scheduling is to achieve maximal concurrency for task execution within a program
  - Maximize resource utilization and minimize inter-process communication
- If there are no precedence constraints except the need for communication among processes, the applications can be better modeled by the communication process model
  - Using undirected graph:

$$G = (V, E)$$



Communication process model
Six processes with communication links and costs shown

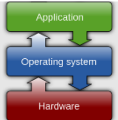


Stone's Algorithm - Assumptions

- Process execution and communication are considered in similar ways so cost (objective function) includes both contributions:

$$cost(G,P) = \sum_{j \in V(G)} e_j(p_i) + \sum_{(i,j) \in E(G)} c_{i,j}(p_i, p_j)$$

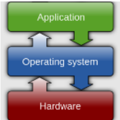
- The system consists of heterogeneous (not-identical) processors
  - Execution depends on the processor to which a process is assigned
  - Execution cost $e_j(p_i)$ for each process is known for all participating processors
- Communication cost between each pair of processes $c_{i,j}(p_i, p_j)$ is known
  - Inter-process communication incurs negligible (zero) cost if both the processes are in the same processor
- Module allocation problem first formulated by Stone – finding an optimal assignment of **m** process modules to **P** processors with respect to the cost function



Stone's Algorithm for Two Processors

- Stone's algorithm minimizes the total execution and communication **cost(G, P)** by properly allocating the **P** processors among **m** processes
- For P = 2, Stone suggested an efficient polynomial-time solution:
  - G = (V,E) : communication process model
  - *A* and *B* : two heterogeneous processors
  - $w_A(u)$ and $w_B(u)$: cost of executing process *u* on *A* and *B*, respectively
  - c(u, v) : communication cost between processes *u* and *v* if they are allocated different processors
  - S : set of processes to be executed on processor *A*
  - V-S : set of process to be executed on processor *B*
- Stone's algorithm computes S such that the cost function is minimized

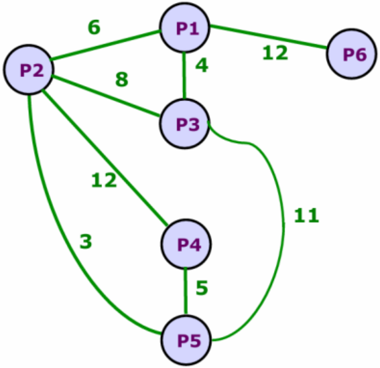
$$cost = \sum_{u \in S} w_A(u) + \sum_{v \in (V-S)} w_B(v) + \sum_{u \in S, v \in (V-S)} c(u,v)$$



Communication Process Model Example

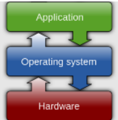
Process	Cost on *A*	Cost on *B*
1	5	10
2	2	∞
3	4	4
4	6	3
5	5	2
6	∞	4

Computation cost



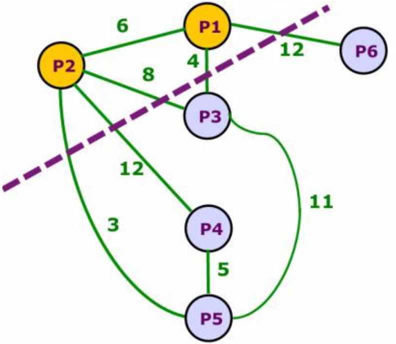
communication cost

A program consisting of six processes (1 through 6) to be allocated on two (non- identical) processors *A* and *B* for minimizing the total computation and communication cost



Partitioning Communication Graph

Process	Cost on A	Cost on B
1	5	10
2	2	∞
3	4	4
4	6	3
5	5	2
6	∞	4



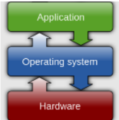
Partition the graph by drawing a line that cuts through some edges. This results in two disjoint graphs, one for each processor

The set of edges removed by the cut is called a *cut set*. Its cost is the sum of weights of the edges, which represents the total interprocess communication cost between ***A*** and ***B***

If processes are allocated to processor ***A*** and the rest to the processor ***B***:

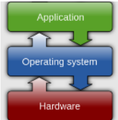
$$cost = (5+2) + (4+3+2+4) + (12+4+8+12+3) = 7 + 13 + 39 = 59$$

Compute cost on A *Compute cost on B* *Comm. cost between A and B*



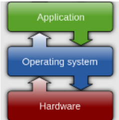
Commodity Flow Problem

- Stone's algorithm reduces the scheduling problem to the commodity-flow problem described below:
- Let **G = (V, E)** be a graph with two special nodes **s** (source) and **d** (destination/sink). Each edge has a maximum capacity to carry some commodity. What is the maximum amount of the commodity that can be carried from the source to the sink?
- Let **S** be a subset of **V** such that the source is in **S** and the sink is in **D** (=V-S). A set of edges (say **C**) with one end in **S** and the other end in **D** is called a cut set and the sum of the capacities of the edges in **C** is called the weight of the cut set
  - Cut is a node partition **(S, D)** such that **s** is in **S** and **d** is in **D**
  - Cut set separates the source and destination in the graph
- Given a commodity graph, the optimization problem is to find the maximum flow from source to destination. It can be shown that the maximum-flow is equal to the minimum possible cut-weight
  - This is called max-flow, min-cut theorem



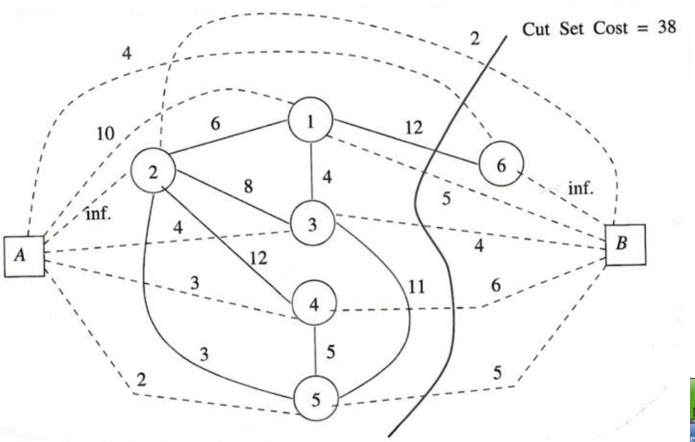
Minimum-cost Cut

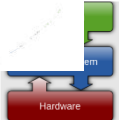
- Given communication process graph **G = (V, E)**, construct a new graph **G'= (V',E')** by adding two new nodes **s** (source, corresponding to processor **A**) and **d** (destination, corresponding to processor **B**). For every node **u** in **G**, add the edges *(**s**, u)* and *(**d**, u)* in G'. The weights (capacities) of the new edges will be $\mathbf{w_B(u)}$ and $\mathbf{w_A(u)}$, respectively
  - $w_A(u)$: execution/computation cost of process **u** on processor **B**
  - $w_B(u)$: execution/computation cost of process **u** on processor **A**
- A cut in **G'** (through the dashed-lines edges) gives a processor allocation for the job/task
- Only consider cuts that separate the processor nodes (A, B). The weight of the cut set is the sum of computation and communication costs
  - Therefore, if we compute the **max-flow** on G', the corresponding **min-cut** gives the best processor allocation



Commodity Flow Problem

Process	Cost on *A*	Cost on *B*
1	5	10
2	2	∞
3	4	4
4	6	3
5	5	2
6	∞	4

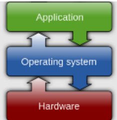




Problem Generalization

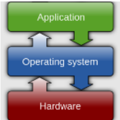
- To generalize the problem beyond two processors, use a repetitive approach using the two-processor algorithm to solve n-processor problem

- Finding a module allocation of **m** processes to **n** processors:
  - The max-flow min-cut algorithm can be applied to a processor $\mathbf{p_i}$ and an imaginary super processor **P** that consists of the remaining processors
  - After some processes have been scheduled to $\mathbf{p_i}$ the same procedure is repeated iteratively on the super processor until all processes are assigned



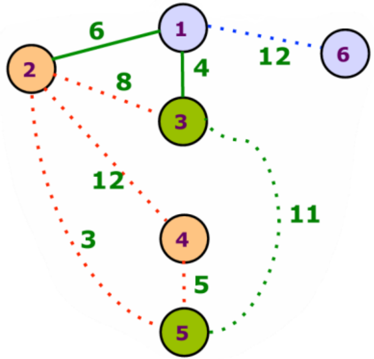
Clustering Processes

- The module allocation problem is complex, requiring heuristic solutions:
  - Heuristic solutions make sense since we generally have approximate information about computation and communication costs
  - Optimal static scheduling has high complexity
- One heuristic approach is to separate the optimization of **computation** and **communication** in two independent phases
  - If communication cost is relatively high, merge processes with higher inter-process interaction into clusters of processes
  - The number of processes in a cluster can be constrained using a threshold on communication cost (**C**) and a threshold on total execution cost (**X**)



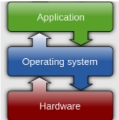
Clustering Example

Process	Cost on *A*	Cost on *B*
1	5	10
2	2	∞
3	4	4
4	6	3
5	5	2
6	∞	4



- With estimated average communication cost ***C* = 9**

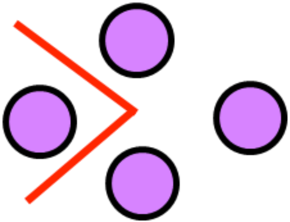
  as threshold, we find three clusters: **(2,4), (1,6), (3,5)** – *12, 11, and 12 are larger than 9*
- Must assign (2,4) to *A* whereas (1,6) to *B*. Cluster (3,5) can be mapped to either *A* or *B*.
  - If assigned to *A*, *total cost* = computation cost of 17 on *A* + computation cost of 14 on *B* + communication cost of 10 between *A* and *B* (overall **41)**
  - If cluster (3,5) assigned to *B*, communication cost becomes too high (6+8+3+5 = **22**)

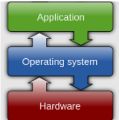


Dynamic Scheduling

Dynamic Scheduling

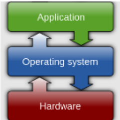
- Prior knowledge of processes (about their execution times and communication behaviors) is not realistic for most distributed applications
- Need an ad hoc scheduling strategy that is adaptive (dynamic) and allows its assignment decision to be made locally (decentralized)
  - Static scheduling is non-adaptive and tends to be centralized
- Using disjoint process model:
  - Ignoring the effect of the interdependency among processes as we do not know how these processes interact with each other
- Objectives/goals:
  - Maximize the utilization of the system – considering throughput and completion time
  - Provide fairness to the user processes – giving priority to a user's process if the user has a lesser share of the resources





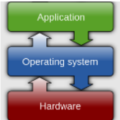
Load Sharing and Balancing

- Performance can be enhanced if processes are free to be redistributed or moved around among the processors/nodes/sites in the system
- Distribute workloads among all available processors to avoid having idle processors as much as possible – load sharing
  - An arriving (new) process can be assigned to the processor that has the shortest waiting queue to reduce processor idling
- Distribute workloads among all nodes as evenly as possible to keep the workload balanced among all nodes of the system – load balancing
  - Needs to move processes from heavily loaded nodes to idle or lightly loaded nodes dynamically
  - Improves performance and achieves a sort of fairness in terms of equal workload for each process
  - Reduces the overall averaging turnaround time of processes



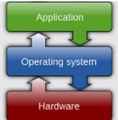
Processor Affinity

- A process has an affinity for the processor on which it is currently running – processor affinity
- Processor affinity makes more sense in SMP (symmetric multiprocessing) systems because of cache memory (locality)
  - If a running process migrates to another processor, the contents of the cache memory must be invalidated for the first processor, and the cache for second processor must be repopulated
  - To avoid high overhead, SMP systems try to keep a process running on the same processor as much as possible
- The process migration overhead is much worse in a distributed system
  - Complete state information of migrating process must be passed to a remote site for its execution there
- Processor affinity counteracts the benefits of load balancing



Centralized Approach

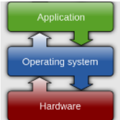
- Designate a controller process that maintains the information about the queue size of each processor
  - Processes arrive and depart from the system asynchronously
  - Controller is responsible for transferring a process from one site to other site: Migrating process from a longer queue to a shorter queue dynamically
- An arriving process makes a request to the coordinator for the assignment to a processor.
- The controller schedules the process to the processor that has the shortest waiting queue
- To update the queue size information, each processor must inform the controller whenever a process completes its execution – overhead
- To remove the centralized controller, the process transfer must be initiated by either a sender or receiver or both



Sender-initiated Algorithms

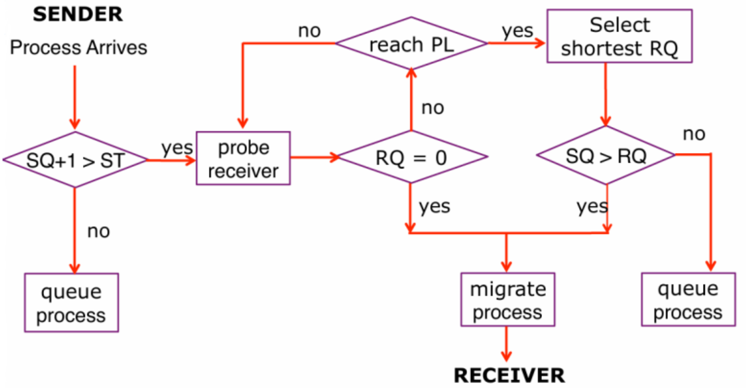
- A sender-initiated algorithm is activated by a sender process that wishes to off-load some of its computation (push model)
  - Facilitates migration of process from a heavily loaded sender to a lightly loaded receiver

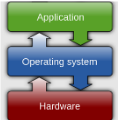
- This process transfer procedure require three basic decisions
  - Transfer policy: When does a node become the sender?
    - When node's queue length (**SQ**) becomes longer than a certain threshold (**ST**)
  - Selection policy: How does the sender choose a process for transfer?
    - A newly arrived process is a good candidate
  - Location policy: Which node should be the target receiver?
    - Randomly select a receiver node
    - Probing receiver: poll a certain number of nodes (probe limit **PL**) to find an idle node or a node with the smallest queue length



Flow Chart of Sender-initiated Algorithm

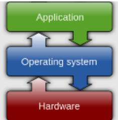
- **SQ**, **ST** and **PL** are the sender's queue length, threshold, and probe limit, respectively
- **RQ** is the queue length of a polled receiver





Sender-initiated Algorithm (Cont.)

- Sender-initiate load sharing algorithm is push model or push migration, where processes are pushed from one node to the others
  - Pushing (or moving) processes from the overloaded to idle or less-busy nodes
- A sender-initiated algorithm incurs additional communication overhead because of probing of receivers and migration of processes
- In an already heavily loaded system, if many (or all) nodes initiate the algorithm simultaneously, a ping-pong effect among senders trying to off- load processes fruitlessly can occur
  - The algorithm becomes unstable at heavy system load due to high communication overhead
- The algorithm perform very well in a lightly loaded system since it is easy to find a receiver

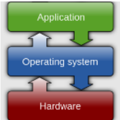


Receiver-initiated Algorithms

- A receiver-initiated algorithm is activated by a receiver process to pull a process from others to its site for execution – pull model or pull migration
  - Pulling waiting process from a busy processor/node

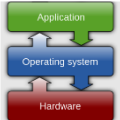
- Three similar basic decisions include
  - Transfer policy – activate pull operation when the queue length (**RQ**) falls below a certain threshold (**RT**)
  - Selection policy – require preemption since the processes at the sender site might have already been running
  - Location policy – a probing strategy to identify a heavily loaded sender

- Receiver-initiated algorithms are more stable than the sender-initiated ones
  - More effective at higher system load because a sender can be found easily and migration communication overhead is low. At very high load, the algorithm becomes less active (so no unstable problem)



Sender/Receiver-initiated Approach

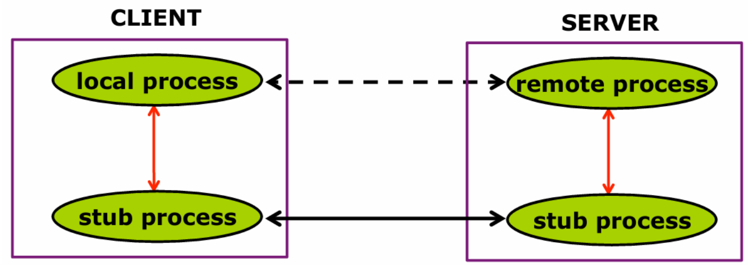
- It makes sense to combine two algorithms into one so that a node can dynamically play the role of either a sender or a receiver
  - Node can activate the sender-initiated algorithms when its queue size exceeds the threshold **ST** and can enable the receiver-initiated algorithms when its queue size falls below the threshold **RT**
- The decision of which algorithm to use can be based on the estimated system load information – an adaptive approach
- Senders rendezvous with receivers
  - A registration service can be used to match a sender with a receiver --
  - Can even serve as a trader for best matching based on some price (e.g., computational cost)

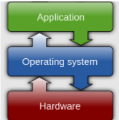


Distributed Process Implementation

Distributed Process Implementation

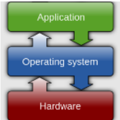
- Load sharing and balancing in distributed system require the activation of process execution at a remote site
- Creating a remote process using the client/server model
  - Front-end stub processes facilitate the creation and communication between processes on different machines
  - Stub processes serve as a logical link between local and remote processes





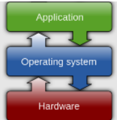
Different Remote Applications

- Based on how request messages are interpreted, different application scenarios can arise:
  - Remote service: The message is interpreted as a request for a known service at the remote site
  - Remote execution: The messages contain a program to be executed at the remote site
  - Process migration: The messages represent a process being migrated to the remote site for continuing execution



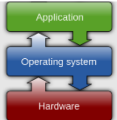
Remote Service

- The request message is interpreted as a request for a known service at the remote site
- Primary application is resource sharing in distributed systems
  - Sharing of file systems, peripherals, and processing capabilities
  - Remote service views the computation environment as accessing remote resources
  - Whereas remote execution and process migration views it as remote host
- Request message can be generated at three different software levels:
  - As remote procedure calls at the language level
  - As remote commands at the operating system level
  - As interpretive messages at the application level



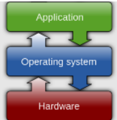
Remote Execution

- Remote execution (or dynamic task placement) is used for off-loading computation (load sharing):
  - The message sent from a client to a server is a client program to be executed at the server
  - In essence, it is the spawning of a process at a remote host
- Remote execution is difficult to implement:
  - Load sharing algorithm – process (or) servers are responsible for maintaining load distribution, negotiating remote host, invoking remote operation, and creating stub processes for linking clients and servers
  - Location independence – each remote process is represented by an agent process at the originating host. Parent/child relationship is preserved
  - System heterogeneity – need to recompile the program or use Java program execution or XDR data transfer
  - Protection and security – a double-edge sword. A foreign code image can be a Trojan horse

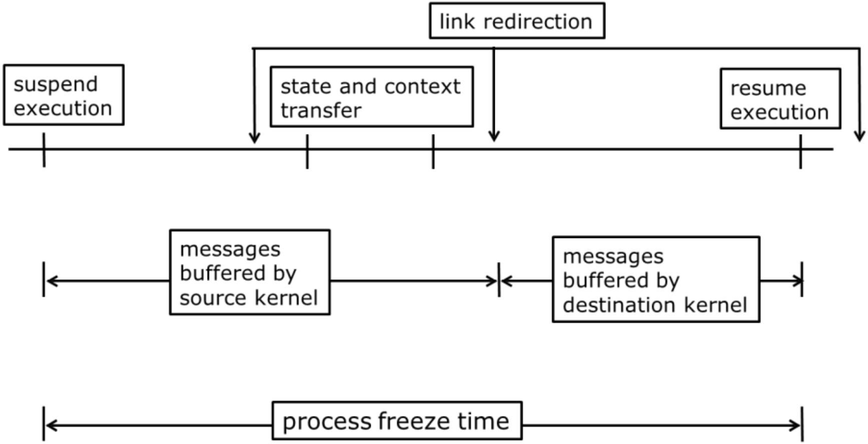


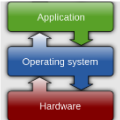
Process Migration

- Extend load-sharing model further to allow a remote execution to be preempted and moved to another remote host
  - In effect, a process can migrate from host to host dynamically
- A process migration facility needs to locate and negotiate a remote host, transfer the code image, and initialize the remote execution
- Since the target process is preempted, its state information
  - Computation state – information necessary to save and restore process at remote site, similar to conventional context switching
  - Communication state – status of process communication links and the messages in transit, difficult to handle
- Two key components of process migration procedure:
  - Link redirection and message forwarding
  - State and context transfer



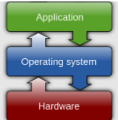
Handling Process Migration

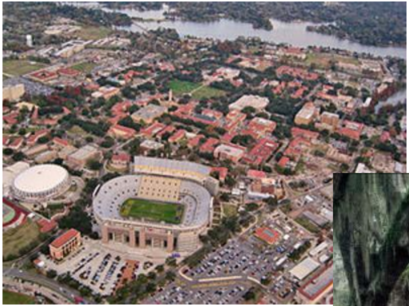




Process Frozen Time

- Freeze (suspend) migrating process after remote host identified
- Transfer the state and context (code image) of the process to remote host
- Perform link redirection at different stages of migration
  - Explicit update requests for link tables of all communicating processes
  - The time of link update affects how process's received messages are forwarded
- Early messages are buffered by source kernel and are transferred together with the context or forwarded later
- Late messages (received after link update) are handled by destination kernel
- Freeze time is process migration overhead
  - Context transfer, link redirection, and message forwarding can proceed concurrently – some overlapping can reduce the overhead
  - To start remote execution requires the transfer of the process's computation state information and some initial codes (i.e., some blocks/pages needed)
  - Link redirection and message forwarding can wait until remote execution begins





LSU CENTER FOR COMPUTATION & TECHNOLOGY

STE||AR GROUP