CSE221 Lecture 1

Aronya Baksy

September 2024

1 THE multiprogramming OS: Edsger W Dijkstra

1.1 Primary goals of the system

- Process continuously a smooth flow of programs
- Reduction in turn around time for programs executing in a short time
- Efficient use of peripheral devices
- Automatic control of the backing store (disk) to be combined with economic CPU usage
- Be economically feasible for simple programs (not requiring excessive power or memory/disk capacity)
- To be easy to debug and verify (verifying soundness of design only, not performance metrics)
- It is **not** a multi-user system, can't have multiple users access it at the same time

1.2 System Specs

- Memory: access time of 2.5 μs, 32K addressable memory
- Drum: 512K words, 1024 words per track, access time 40ms
- Indirect addressing mechanism
- sound system for commanding peripherals/controlling interrupts
- potentially large number of "low capacity channels" - 3 paper tape readers at 1000 chars/sec.

1.3 Storage Allocation

- Distinction is made between *core pages* in memory and *drum pages* on disk
- Programs think in terms of *segments*. Each segment can span multiple real pages (on disk or in memory)
- Segments have their own identification numbers, segment ID numbers are much larger than no. of pages in mem/disk.
- Segment ID gives fast access to a "segment variable" in core which contains the real page address
- if a segment resides in a core page needs to be dumped to a drum (disk), there is no need to return the segment to the place where it initially came from. **The minimum latency time to access the drum is used to store the page**
- Programs do not need to occupy consecutive drum pages

1.4 Processor Allocation

- System is arranged in terms of **sequential processes** (in contrast with electrical circuits that compute in one single step) co-operating with each other using explicit **mutual synchronization** statements
- **Benefits**: co-ordination between processes can be established using discrete reasoning, system is independent of number of processors available (as long as each one can context-switch)

1.5 System Hierarchy

- **Level 0**: responsible for *processor allocation*, processes clock interrupts and prevents one process from *monopolizing* the system
- **Level 1**: *Segment controller* process, synchronized to the disk interrupt, handles bookkeeping tasks for converting segment IDs to page numbers (”automated backing store”)
- **Level 2**: *Message interpreter* takes care of peripheral keyboard input via an interrupt. Output of the device is sent to the teleprinter.
  - A process who opens a conversation identifies itself - the operator must also be able to identify a running process if they want to open a conversation
  - above level 2, each process sees itself as having its own private console - however, it is merely an abstract achieved via mutual synchronization.
- **Level 3**: Handling I/O for peripherals, sequential processes associating with buffering input streams and unbuffering output streams
- **Level 4**: user processes
- **Level 5**: the operator

1.6 Synchronization Primitive - the Semaphore

- This paper introduced the concept of the ”semaphore” - a synchronization primitive
- Two operations - wait (referred to as P) and post (or signal, referred to as V). All read/writes on the semaphore are **atomic**
- P() waits until the semaphore value is > 0 then decrements the value of the variable
- V() increments the value

1.6.1 Mutual Exclusion

- The paper defines the *critical section* as the section of a program needed to unambiguously inspect and modify the state variables that describe the current state of the system
- The paper defines both binary semaphores (using them just like mutex locks) and counting semaphores
- The paper also defines *private semaphores* which are used for co-ordinating between processes.
- These are initialized to 0 unlike mutexes described above
  - No other process performs a P op on a process’ private semaphore
  - Processes waiting on values of state vars for their progress can conditionally run `V(pvtsem)`

```
P(mutex);
// inspect and modify state vars
// including conditional V(pvt);
V(mutex);
P(pvt);
```

Process waiting for values of state vars

```
P(mutex);
// inspect and modify state variables
// including V(pvt) on other processes
V(mutex);
```

Unblocking waiting processes

1.7 Proving Harmonious Co-Operation of Processes

- Djikstra proves the harmonious co-operation of all the sequential processes in the system in the following stages:
  1. A single initial task cannot give rise to infinitely many tasks as a task can only generate sub-tasks for processes in levels below it. This implies that number of processes running is finite always
  2. It is proved that there cannot be a situation where all processes have returned to a ”homing” position while there is a generated and unaccepted task still present.
  3. There are no circular waits in the system, hence all process will complete (”return to a homing position”) after being intialized.

2 The Nucleus of a Multiprogramming System - PB Hansen

Developed for the RC 4000 minicomputer built by A/S Regnecentralen in Denmark.

2.1 Primary Goals

- Instead of building a system to satisfy specific needs (real-time/priority sched., batch processing, priority access), be *flexible* in choice of system type
- Build an *extendable* **nucleus** that supports multi-programming and can be extended with an OS on top
- Unlike Dijkstra's paper, less focus on implementation, installation here and instead more on philosophy and structure
- Use *small number of clean abstractions*: process and message

2.2 Implementation Notes

- 24 bit system, typical instruction exec takes 4 μs
- *send msg/answer* operations take around 0.6 ms while the corresponding *wait* operations take around 0.4 ms
- Create/stop processs takes $<$ 5ms while start/remove process takes 26-30ms. This is because of storage protections in RC4000 hardware which requwire setting protection bits in every word

2.3 Components and Defintions

- **Internal process**: execution of one or more interruptable programs in a given storage area, identifiable by a name
- **Peripheral device**: storage medium connected to the data channel identified by a *device number*.
- **Document**: collection of data stored on external media
- **External process**: abstraction of a physical device as a process
- **System nucleus** :
  - Control Multiprogramming and communication between internal and external processes
  - Handle interrupt response
  - Complete control of *I/O , storage protection*

2.4 Process Communication

- Happens through *message buffering*.
- Advantage over sempahores - avoid dealing with malicious processes that don't use semaphores well, hence avoiding deadlocks
- System nucleus administers a common pool of **message buffers**, and a **message queue per process**
- Four operations for communication:
  1. *send message* - copies message into first available buffer and delivers it in the queue of the receiver
  2. *wait answer* - delays the requesting process until an answer arrives in a given buffer.
  3. *send answer* - copies answer into a buffer of a message which was received, pushing it back to the original sender.
  4. *wait message* - delays the requesting process until a message arrives in its queue
- Processes can run unaware of each other until messages are sent.
- Each buffer also contains ID of sender and receiver, verify this every time a process sends/waits on a message buffer
- Limit on the number of messages a process can send simultaneously to avoid buffer space being filled up

2.5 External Processes

- *send message* and *wait answer* are used for communication between internal and external processes.
- System nucleus contains code for each type of external process to interpret msgs from internal processes and initiate I/O on the device specificed by the msg
- Ext processes created on request by int processes. Creation is simply assignment of a name to a peropheral device

2.6 Internal Processes

- Created by a parent process assigning a name to a contiguous storage area selected by the parent process within the parent's own area.
- Parent process loads a program into the storage area of the child and *starts* it
- The system nucleus can *stop* a process on request from its parent after waiting for all I/O operations to complete on that process
- Parent can *remove* a child process to free up its storage area
- Program loading and swapping are **not** part of the nucleus. Only primitives for loading and stopping are. Scheduling algorithm choice is left entirely to processes

2.7 Process Hierarchy

- Processes organized in a tree-like hierarchy where the parent is responsible for operator communication, scheduling and resource allocation for its children processes
- Programs can be written in high-level languages (ALGOL 64 and FORTRAN)
- Operating systems can be replaced dynamically on top of the nucleus