**INTRODUCTION**

An Operating System has two roles:

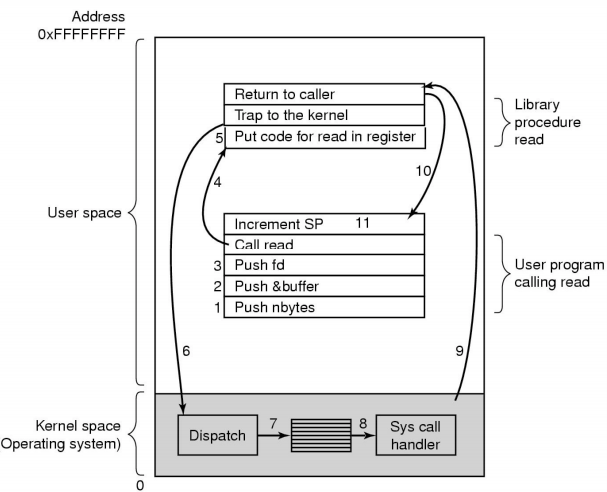
1. The OS is an **abstract machine**: hides hardware details, provide abstraction to programmers and applications.
2. The OS is a **resource manager**: ensure no starvation, progress for all processes, allocation according to desired policy.

**Kernel** portion of an OS runs in privileged mode and mediates access to resources (e.g. CPU, RAM, IO Devices).

**User** portion of an OS interacts with the Kernel via. System Calls

**SYSTEM CALLS**

**System Calls** is a mechanism to cross between the USER LEVEL and KERNEL LEVEL.



*11 steps in making a System Call: read(fd, buffer, nbytes)*

User Program calling read()

|  |  |
| --- | --- |
| #1 #2 #3 | Push parameters on the stack. |
| #4 | Invoke the system call read(). |

Library procedure read()

|  |  |
| --- | --- |
| #5 #6 | Put code for read() on the register so the OS knows what to look at + **Trap** (switch) to the kernel. |
| #7 | **Dispatcher/Scheduler** receives control in kernel mode as a result of the trap, then performs a context switch. Has to choose a **Ready Process** to run from the **Ready Queue**.  **Context Switching**: Dispatcher saves state of previous process that was running then loads the initial/previously saved state of the new process. |

Kernel space actions.

|  |  |
| --- | --- |
| #8 | **Handler** saves register values onto the Kernel Stack + performs validations.  It looks at the **System Call Table** (map of Syscall numbers and appropriate Syscall) to invoke the correct system call function.  After performing the requested actions, the handler restores register values from the Kernel Stack and places the System Call return value on the User Stack. |
| #9 | Process goes back into User Mode at the saved return location. If return value indicated an error, then **errno** (global var) is set that indicates the status of execution. |

**PROCESSES & THREADS**

**Processes** are the execution of an individual program.

**Dispatcher/Scheduler** gives control of the CPU to the process selected using the **Short-Term Scheduler**.

|  |  |
| --- | --- |
| Events that cause Process Creation | Conditions which Terminate Processes |
| * System initialisation: Foreground + Background processes * Execution of process creation Syscall: i.e. new login shell from SSH. * User request to create a new process. * Initiation of a batch job. | * Normal Exit: Voluntary * Error Exit: Voluntary * Fatal Error: Involuntary * Killed by another process: Involuntary |

**Process Control Block (PCB) / Context Block** stores all info needed to keep track of a process. Identified by a **PID (process ID)**

**Process Table** is formed from all PCB’s. Facilitates context switching, scheduling and other activities.

**Thread** is a unit of execution that belongs to a program.

Benefits of threads:

* Simpler to program than a state machine.
* Less resources are needed with them than a complete process.
  + Threads are cheaper to create/destroy + Threads share resources (especially memory) between them.
* Performance: Threads waiting for I/O can be overlapped with computing threads (interchangeable).
  + However, no performance improvement occurs if threads are **Compute Bound** (limited by a single CPU system)
* Take advantage of parallelism available on machines that have multiple processors.

**CONCURRENCY & SYNCHRONISATION PRIMITIVES**

**Race Condition**: Occurs when > 2 threads access shared data and try to change it at the same time.

* Thread Scheduling may lead to un-ordered swapping of threads at any time = un-ordered access to a shared resource.

**Critical Region**: Region of code where the shared resources are being accessed.

* A solution to the Critical Region Problem must satisfy the following requirements:

**MUTUAL EXCLUSION**: No two processes are simultaneously in a Critical Region.

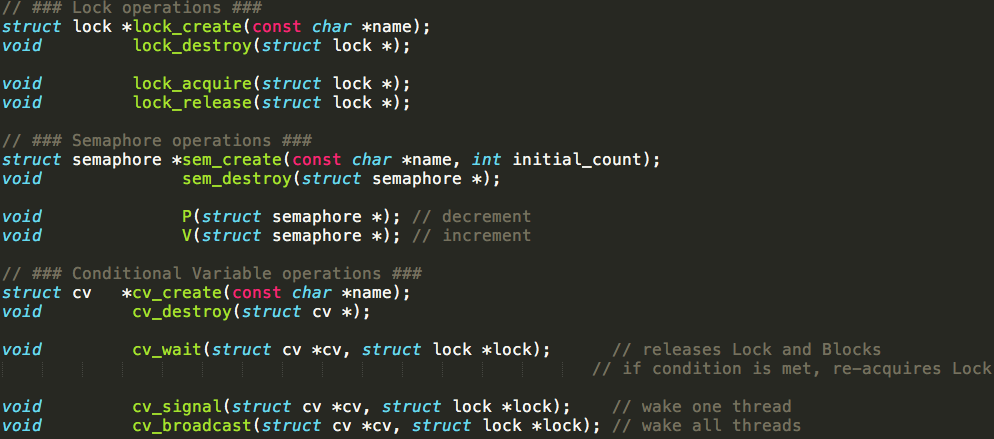
**PROGRESS**: No process running outside the C.R may block another interested process from entering when the region is free.

**BOUNDED WAITING**: There must be a bound on the no. of times other processes can enter their C.R.

**Bounded Waiting** prevents **Starvation**:

* Starvation occurs when a process cannot attain a resource because the scheduler always allocates resources to other processes, denying the starved process from progressing. Solved by FIFO policy (Queues).

Synchronisation Primitives & Operations



**Semaphore**: used to control access to a shared resource by multiple processes.

* P(Semaphore S): [wait] semaphore value--. If value < 0, block process and put in process queue.

V(Semaphore S) : [signal] semaphore value++. Transfers blocked processes from the process queue to the ready queue.

**Mutex**: similar to a Semaphore (semaphore value = 1) except only the thread that holds the Mutex lock may unlock it.

**Monitor:** a group of procedures, variables, data types grouped into a module, where only one thread can be inside the monitor at any time. When a thread calls a monitor procedure with an existing thread inside, it is queued and sleeps until the current thread exits the monitor.

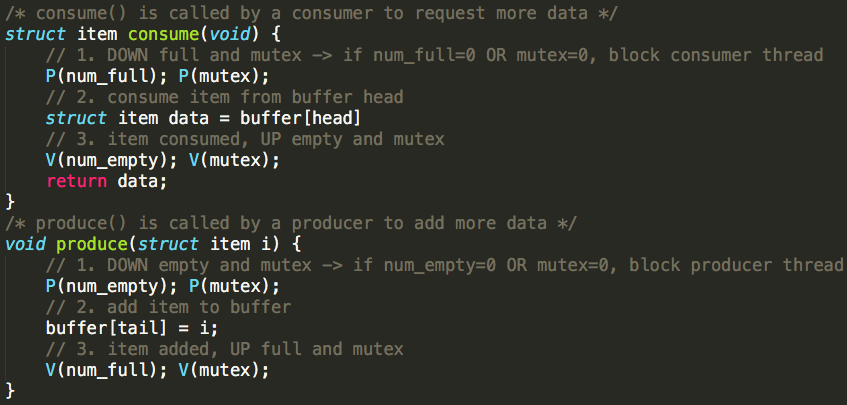
* Consists of a MUTEX + CONDITIONAL VARIABLE.

**Conditional Variables**: provides a queue for threads waiting for a resource, to wait for a conditi

* While a thread is waiting on a conditional variable, it does not occupy the Monitor so other threads may enter the Monitor to change its state.
* Has both Signal (wake one thread) and Broadcast (wake all threads) operations.

**Producer-Consumer Problem (Bounded-Buffer Problem)**

* Producer + Consumer processes who share a common, fixed-size buffer.
* Producer’s job is to generate data (add to buffer) + Consumer’s job is to consume data (remove from buffer) 1 by 1.



Problem:

Make sure the Producer won’t add to a full buffer + Consumer won’t take away from an empty buffer.

Solution:

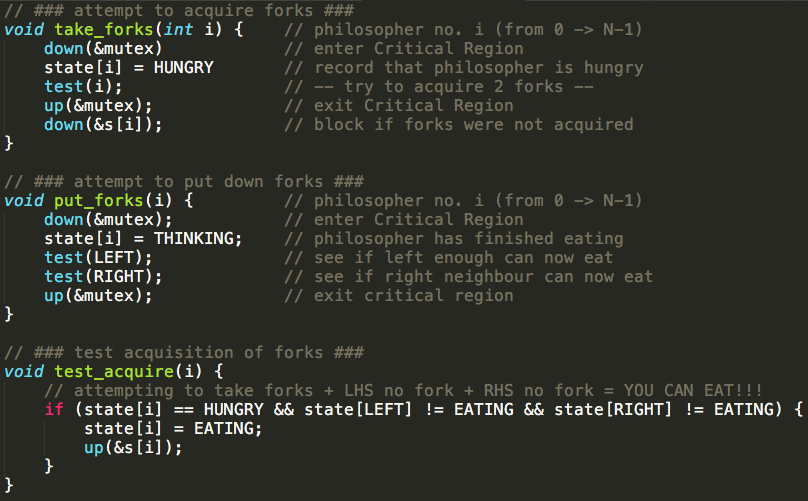
2 semaphores:

* num\_full
* num\_empty

1 mutex object

**Dining-Philosopher’s Problem**:

* K philosophers seated in a circle with one chopstick between each pair of philosophers.
* A philosopher may eat if he can pick up two chopsticks next to him.
* Each chopstick may only be used by 1 person at a time.



Solution:

1 semaphore array:

* One semaphore for each chopstick to control the behaviour of each person

1 mutex object

**DEADLOCKS**

**Deadlocks**: A set of processes is deadlocked if each process in the set is waiting for an event (release of a held resource) that only another process in the set can cause.

**Four conditions of Deadlock**:

1. **MUTUAL EXCLUSION**: Several processes cannot simultaneously share a single resource.  
   (read-only resources do not require this condition therefore can’t be involved in a deadlock)
2. **HOLD AND WAIT**: Hold the resource I have + wait for the resource that I don’t have yet.
3. **NO PRE-EMPTION**: Resources cannot be taken away pre-emptively by another process or by the OS.
4. **CIRCULAR WAIT**: A chain of processes where each process is waiting for a resource that is held by another process.

**Strategies for dealing with Deadlock**:

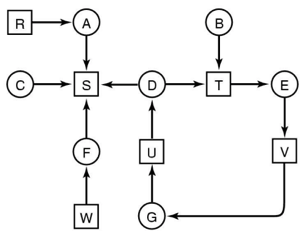
**THE OSTRITCH ALGORITHM**: ignoring problems on the basis that they are extremely rare to occur.

* Reasonable if Deadlocks rarely occur and the cost of prevention is high. (Convenience vs. Correctness trade-off)

**DEADLOCK PREVENTION**: prevent one of the four Deadlock conditions.

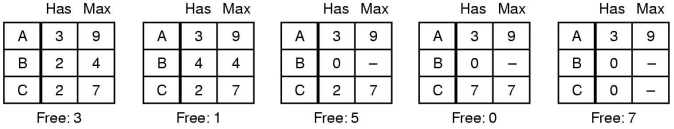
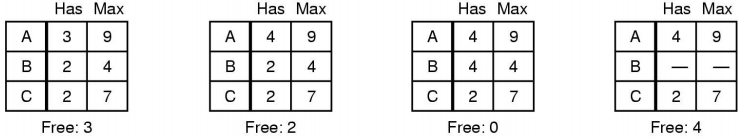
* #1 Mutual Exclusion: NOT FEASIBLE. Some devices/resources cannot be made shareable e.g. Printer.
* #2 Hold-and-Wait: FEASIBLE. Require processes to request ALL resources before starting execution.
  + Not always possible to know all the required resources at the start of execution.
  + Variation of the strategy: process gives up all its resources if it blocks another process, then immediately request all resources again.
  + ^**LiveLock** can occur: set of processes that aren’t blocked, run regularly but never make any progress.
* #3 No Pre-emption: NOT FEASIBLE. Can’t take resources away from a printer halfway through printing.
* #4 Circular Wait Condition: FEASIBLE/BEST. **Resource Ordering** is a common technique used in practice.

*Anytime a process requests a resource, it has to request it in a global (applying to all processes) specified order.*

**DEADLOCK DETECTION & RECOVERY**: let deadlocks happen and come up with a recovery mechanism.

* Resources w/ single unit: look for cycles within the graph, denoting a deadlock. --------------------->
* Resources w/ multiple units: use a detection algorithm
* STEP 1: Mark rows/processes which are EQUAL or GREATER THAN what is available
* STEP 2: Update resources + repeat process above
* STEP 3: If all rows/processes can be marked off, the system is NOT deadlocked.
* Deadlock recovery: recovery is performed through killing one of the processes in the deadlock.

**DEADLOCK AVOIDANCE**: Avoid deadlocks by allocating resources only if the resulting state = SAFE STATE

* Safe State: A system can allocate all resources to processes (up to their max) without entering into a deadlock.
* Unsafe State: Processes are not necessarily deadlocked, but we cannot guarantee that they will not deadlock/complete.
* **Banker’s Algorithm**: **Current Allocated (C) + Resources Available (A) = Resources in Existence (E)**
  + Keep the bank in a “SAFE STATE” where all customers can request to borrow up to their limit simultaneously.
  + Customers wishing to borrow such that the bank would enter an UNSAFE STATE must wait until someone repays their loan such that the transaction becomes safe.
  + Banker’s Algo is not commonly used as it is difficult to determine the amount of resources required in advance.
  + Algorithm:

1. Calculate Available (A) = Existence (E) – Current Allocated (C)
2. Find a row/process in the Requested matrix where Requested <= Available
3. IF row/process exists, completion is possible | ELSE deadlock is possible, implying INITIAL STATE = UNSAFE.
4. Selected row/process acquires available resources -> execute -> terminate -> return resources back to Available (A)
5. Repeat steps until all processes have successfully reached termination, implying INITIAL STATE = SAFE.

**PROCESSES & THREADS: Implementation trade-offs @User or @Kernel level + Context Switching**

Implementing Threads at the User-Level: **User-Level Thread Control Block (TCB) + Ready Queue + Blocked Queue + Dispatcher**

* Kernel/OS has no knowledge of the threads (it only sees a single process)
* Thread management methods (created, exit, yield, wait) are implemented in a runtime support library.

Implementing Threads at the Kernel-Level: **Kernel-Level Thread Control Block (TCB)**

* A subset of information (related to execution context) in a traditional Process Control Block (PCB).
* TCB’s have a related PCB of the process that the thread lives on.
* Thread management methods (create() wait() exit()) are implemented as system calls.

PROS and CONS of each approach:

|  |  |  |
| --- | --- | --- |
|  | @USER-LEVEL | @KERNEL-LEVEL |
| PROS | * Context Switching + Threat Management is faster than @Kern (trap() to Kernel and back to User not required) * Programmer can choose the scheduling / Dispatcher algorithm for each application. * Can be implemented on any OS. * Can support huge numbers of threads. -> Uses application virtual memory | * **Pre-emptive Multithreading/Scheduling**: execution of a thread can be interrupted by another thread / externally. * **Parallelism**:   -> Can overlap blocking I/O with computation.  -> Can take advantage of a multi-processor |
| CONS | * Threads have to yield() manually (no timer-interrupt delivery)   -> **Co-operative Multithreading/Scheduling**: a model where once a thread is given control it continues to run until it explicitly yields control or blocks *(doesn’t respond to external events)*  -> A single poorly implemented user-thread can monopolise the available CPU time.   * Does not take advantage of multiple CPU’s. * If a thread makes a blocking system call (a syscall which causes a thread to wait until the Syscall returns), the process and all internal threads will block i.e. a “blanket block” -> Can’t overlap I/O with computation. | * Thread creation, destruction, blocking and unblocking threads require Kernel entry and exit = more expensive operations than User-approach. |

**Context Switch** is the process of storing the state of a thread/process, so it can be restored and resume execution later.

* A Context Switch occurs when the OS is invoked on a Syscall, Exception, Interrupt.

Example Context Switch Scenario:

1. (USER MODE) SP is pointing to User Stack. Exception/Syscall/Interrupt occurs and we switch to Kernel Stack
2. We push a trap frame on the stack (saves state, including user-level Program Counter and Stack Pointer).
3. ‘C’ code is executed to process the Exception/Syscall/Interrupt. “C activation stack” builds up.
4. The Kernel decides to perform a context switch.
   1. It chooses a Target Thread/Process + pushes remaining Kernel context onto the stack.
5. We save the current SP in the Process Control Block (or Thread Control Block) and load the SP of the Target Thread.
   1. We have now switched contexts.
6. Load the Target Thread’s previous context 🡪 return to C / User Mode.

**COMPUTER HARDWARE: Memory, Hierarchy & Caching**

**Registers** (FASTEST): holding results / variables.

**Cache**: hardware to hold memory which is being accessed repeatedly.

-> Cached for fast access.

-> L1 Cache: accessed without any delay

-> L2 Cache: takes more clock cycles than L1

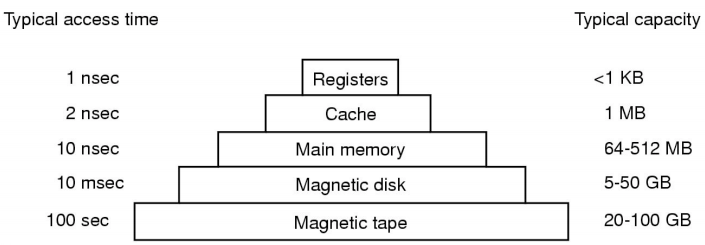
-> L3 Cache: takes more clock cycles than L2

**Main Memory / RAM**: hardware component

**Hard Disk**: data is kept permanently here. Not directly accessed by CPU

**Magnetic Tape**: used for backing up large data. Mount/unmounted

**Computer Hardware Hierarchy**



**Word Transfer**: Word-by-word transfer between registers and cache.

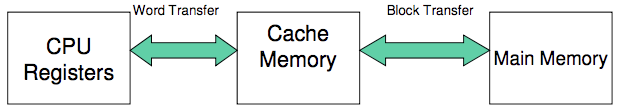
**Block Transfer**: Fixed-size block of data transferred between cache and main memory.

**Cache Hit**: CPU finds requested mem address in Cache.

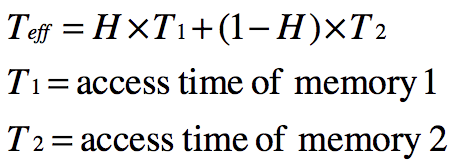
**Cache Miss**: CPU can’t find requested mem address in Cache.

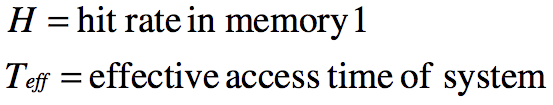
*Cache allocates a new entry, copies data from the requested address from Main Memory then the request is fulfilled from the contents of the cache.*

**Cache Transfers**

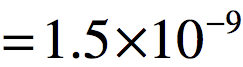
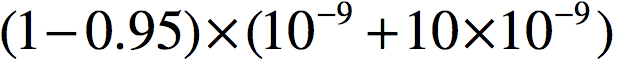
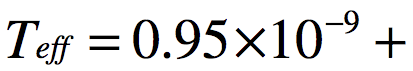


**Cache Effective Access Time (EXAMPLE #1):**





EXAMPLE: Cache memory access time = 1ns | Main memory access time = 10ns | Hit Rate = 95%



**Cache Effective Access Time (EXAMPLE #2):**

USE PAST PAPER EXAMPLE

**Moving Head-Disk Mechanism:** a disk storage device which one or more read/write heads are attached to a moveable arm allows each head to cover many tracks of information.

* Disk read/write is fast, however access time is dominated by need to position/seek the head over the data.
* Improve system performance by avoiding seek time using Main Memory / RAM as a Cache for disk content.

**FILE MANAGEMENT / FILE SYSTEMS**

**Files Introduction**

File Abstraction: Files as Byte Sequence 🡪 Files as Record Sequence 🡪 Files as Tree of Records, indexed by a Key

File Types: Regular files, Directories, Device files (CHARACTER: un-buffered access vs. BLOCK: buffered access)

File Access Types: Sequential Access (r/w from beginning, can only rewind) vs. Random Access (r/w in any order)

File Attributes: Protection (RWX), password, creator, owner, record len, creation time, last accessed + more.

Typical Operations: create(), delete(), open(), close(), read(), write(), append(), seek(), get\_attributes(), rename() + more.

**Files Introduction**

**ONLY PUT USEFUL CONTENT FROM HERE ON**

**MEMORY MANAGEMENT / VIRTUAL MEMORY (Page-Based)**

**Internal Fragmentation vs. External Fragmentation**

**Page Size / Page Table Calculation**

Assume 32bit virtual address space (2^32 = address values), mapped to 4kB size pages.

* 4kB page size = 4,000bytes = **~2^12 bits per page**
* 2^32 / 2^12 = **2^20 no. of pages & no. frames (page size = frame size)**
* 20 bits required for frame number + 1bit (present/absent) + 1bit (modified) + 1bit (reference) + 1bit (prot) + 1bit (caching) = 25 bits = **4 bytes for each entry.**
* Page table size = 4 bytes \* 2^20 pages = 2^2 \* 2^20 = 2^22 bytes = 4,194,304 bytes = **~4MB Page Table size**

**Page Replacement Algorithms**

**INPUT / OUTPUT MANAGEMENT**