CS2106

Introduction to Operating Systems

AY2022/23 Semester 2

Notes by Jonathan Tay

Last updated on April 6, 2023

Contents

Part I

Process Management

1 Process Abstraction

A **process** abstracts the information required to describe and manage the execution of a program.

1.1 Function Invocation

Function invocation presents control flow and data storage challenges:

- jumping to function body, and resuming after the function returns
- passing parameters, and capturing a return value
- allocating local variables

stack memory

Stack memory is a region of memory separate from instruction and data memory that is used to store the following information for function invocations in **stack frames**:

1. return address (PC) of the caller
2. arguments for the parameters of the function
3. local variables used by the function
4. saved **stack pointer (SP)**

Optionally, the stack frame also includes:

5. **frame pointer (FP)**
6. saved registers

Stack frames are **pushed** onto the stack on invocation and **popped** on return.

The **stack pointer (SP)** is a special register which contains the address of the first free location in the stack.

Because SP varies depending on the number of local variables, some processors provide a **frame pointer (FP)** which points to a fixed location in a stack frame.

When all general purpose registers are exhausted (**register spilling**), their values can be written to memory and restored at the end of the function call.

function invocation — setup

If a function `f()` invokes `g()`, then `f()` is the **caller** and `g()` is the **callee**.

- **caller**:
  1. passes arguments by setting registers directly and/or pushes arguments onto the stack
  2. saves the return address on the stack
- **callee**:
  1. saves the old stack pointer
  2. allocates space for local variables
  3. sets SP to the top of the stack

function invocation — teardown

- **callee**:
  1. pushes return value onto the stack (if any)
  2. restores SP to the saved stack pointer
  3. sets PC to the return address
- **caller**:
  1. uses the return result (if any)
  2. resumes execution

Actual implementation of the function call convention depends on hardware, programming language, and compiler — it is not universal.

1.2 Dynamically Allocated Memory

Some data may have a size only known at runtime, which precludes allocation in the data memory.

Some data may have an unknown lifetime (i.e. no deallocation time guarantees) which precludes allocation in the stack memory.

Hence, such dynamic data is allocated (e.g. by `malloc`) in the **heap memory**.

1.3 Process Model

Process IDs (PIDs) uniquely identify processes, each of which may exist in 5 different possible states:

1. **new**: created, or under initialization
2. **ready**: admitted, but waiting to run
3. **running**: being executed by the CPU
4. **blocked**: waiting until an event occurs
5. **terminated**: finished execution, possibly requiring OS cleanup

process control blocks (PCBs)

Every process' registers, memory, PID, and state is stored in a **process control block**.

The kernel manages all PCBs in a **process table**.

context switching

Programs may be interrupted at any time, e.g. by the **scheduler**, and must operate independently of interrupts.

A **context switch** saves the current execution context process in a PCB and loads the context of the next process to be executed.

1.4 System Calls

System calls, or syscalls, are an API for programs (or user-friendly libraries) to call services in the kernel.

Each syscall has a unique **system call number**, which is passed to the kernel via a designated register along with any arguments.

The signature of a syscall in C is `long syscall(long number[, arg1, arg2, ...])`.

These functions require a change from user mode to kernel mode when invoked, via a **TRAP instruction**.

Syscalls are laid out in a table, which a **dispatcher** uses the system call number to find the correct **system call handler**.

The system call handler executes the request, returns control to the library call, and switches back to user mode.

1.5 Exceptions and Interrupts

Machine level instructions can cause ***synchronous* exceptions** during program execution, e.g. division by zero or illegal memory access, which then passes control to an **exception handler**.

External events such as timers or hardware can cause ***asynchronous* interrupts**, which the operating then searches the **interrupt vector table** (IVT) to invokes the correct **interrupt handler**.

The IVT is set up by the OS during boot, and exists in memory (hardware).

As interrupts are asynchronous, they are only checked at specific times; never in the middle of an instruction: instructions are atomic/indivisible.

exception/interrupt handlers

These handlers save the register and CPU state, handle the necessary routine, and then restore the state and return to the interrupted program.

Program execution may resume as though the exception or interrupt never occurred.

1.6 Example: Unix Processes

Every process has an integer process ID (PID). Child processes also remember their parent's PID.

`init` is the ancestor of all other processes, created by the kernel at boot, with PID 1.

Process status can be viewed with the `ps` command.

`fork()`

`fork()` creates a new **child process** which shares the same code and address space as the parent process.

Only the data memory (e.g. variables) of the child process is copied from the parent instead of shared.

Because memory copying is expensive, some systems use **copy-on-write**.

`fork()` returns `0` from the child process and the PID of the child process from the parent process.

`execl()`

`execl()` replaces the current process with a new process specified by the first argument, with optional `NULL`-terminated command-line arguments:

```
int execl(char *path, char *arg, ..., NULL);
```

wait()

`wait()` blocks the parent process until at least one child process terminates, after which it clears the remainder of the child process' resources which are not removed by `exit()`.

`wait()` returns the PID of the terminated child process, while also accepting an optional `int` pointer for the child process' exit status:

```
int wait(int *status);
```

If the parent process terminates before its child processes, the **orphan** child processes are adopted by `init` which will use `wait()` to cleanup.

If the child process terminates with `exit()` but its parent does not call `wait()`, then it becomes a **zombie process**.

2 Inter-process Communication

Two processes P_1 and P_2 need to communicate with each other. There are two ways to do this:

1. **shared memory**:
   P_1 creates a shared memory region which P_2 can attach to. P_1 and P_2 can then read and write to the shared memory region.
2. **message passing**:
   P_1 and P_2 exchange messages, where sending and receiving involve the OS via syscalls.

The OS overhead for message passing is much higher than for shared memory, where the OS is only involved during the setup and teardown.

Both options require some form of synchronization, to avoid **data races**, where the processes read and write to the same memory location at the same time.

Race conditions may result in incorrect behavior.

direct & indirect communication

In a shared memory scheme, communication is always *implicit* — a bystander cannot easily detect that communication is taking place.

However, in message passing, communication is *explicit* — the sender must explicitly name the recipient, or specify a common message storage, usually a shared **mailbox** or **port**.

This means message passing may require the use of finite *kernel memory* to buffer messages.

synchronous & asynchronous message passing

The `receive()` operation is always assumed to be *blocking* — always waiting until a message arrives.

Asynchronous message passing is *non-blocking*, requiring the kernel to buffer messages. In the case of buffer overflow, `send()` may:

- block until the buffer has space, or,
- return an error.

Synchronous message passing is *blocking*, requiring the sender to wait until the receiver has received the message. No buffering is required.

2.1 Example: Unix IPC

shared memory

1. The master program creates the shared memory space via `shmget()` and attaches it via `shmat()`.
2. The worker process only needs to attach to the shared memory space via `shmat()`.
3. Both processes use some space in the shared memory as control values for synchronization.
4. After work is complete, the worker process detaches from the shared memory via `shmdt()`.
5. The master process detaches from the shared memory via `shmdt()` and destroys the shared memory via `shmctl()`.

message passing — pipes

Every process has `stdin`, `stdout`, and `stderr` pipes, which can redirected with <, >, and 2> respectively.

Pipes in a **producer-consumer** relationship are connected with | at the command line.

Pipes are created with the `pipe(fd[2])` syscall in C, which returns two file descriptors, one for reading and one for writing.

The producer must `close()` the read end of the pipe before writing to it, and then `close()` the write end of the pipe after writing to it.

The consumer must `close()` the write end of the pipe before reading from it, and then `close()` the read end of the pipe after reading from it.

signals

Signals are asynchronous notifications sent to a process by the OS or another process.

Processes can define their own signal handlers to handle signals, such as `SIGSEGV`, using the `signal(SIGNAL, *fp)` syscall in C.

The `SIGKILL` and `SIGSTOP` signals are sent by the kernel to a process to terminate it, and cannot be caught, blocked, or ignored.

3 Process Scheduling

To execute multiple processes concurrently, instructions from processes on the same CPU core are interleaved in a process known as **timeslicing**.

The **scheduler** is the part of the OS which makes selects processes to run using **scheduling algorithms** to ensure *fairness* (no starvation) and *full utilization* of the CPU.

There are 2 kinds of scheduling policies:

1. **preemptive**:
   processes are given a fixed time quota, and are interrupted if they do not block or terminate
2. **non-preemptive**:
   processes remain in the running state until they block or terminate *voluntarily*

Processes go through phases of **CPU activity** and **IO activity** and their processing environment falls into 3 categories:

1. **batch processing**:
   no user interaction, need not be responsive
2. **interactive**:
   must have a low and consistent response time
3. **real-time processing**:
   deadlines must be met, and usually periodic

3.1 Batch Processing Scheduling

Batch processing are evaluated on 4 metrics:

- **turnaround time**: finish time - arrival time
- **throughput**: number of tasks finished per unit time
- **makespan**: total time taken to complete all tasks
- **CPU utilization**: percentage of time the CPU is working

first come first serve (FCFS)

Tasks are stoped in a FIFO queue, enqueued when they arrive and dequeued to run until they block or terminate.

Guarantees **no starvation** as the number of tasks is strictly decreasing.

Suffers from the **convoy effect** — a CPU-bound process blocks queuing IO-bound processes, which leads to I/O idling, after unblocking, the IO-bound processes only require low CPU utilization.

A simple reordering of processes in the queue can reduce average waiting time.

shortest job first (SJF)

Tasks are ordered in a priority queue with the task with the least CPU time estimated given the highest priority.

Estimation is typically done using an exponential moving average:

$\text{Predicted}_{n+1} = \alpha \text{ Actual}_n + (1-\alpha) \text{ Predicted}_n$, where α is the weight given based on past history.

Allows **preempting** a running task if a task with a shorter job arrives.

Guarantees smallest average waiting time.

Suffers from possible **starvation**.

shortest remaining time (SRT)

Similar to SJF, except priority is given to the task with the least remaining CPU time.

Allows **preempting** a running task if a task with a shorter remaining time arrives.

Suffers from possible **starvation**.

3.2 Interactive Scheduling

Interactive environments are evaluated by their **response time** and **predictability**, which is the variation in response time.

A timer interrupt (which the OS ensures cannot be intercepted) is raised at fixed intervals and its handler invokes the scheduler.

This divides CPU time into **time quantums**, which are multiples of the timer interrupt interval.

round robin (RR)

RR is the preemptive version of FCFS, where tasks in the FIFO queue are given a fixed time quantum to run, after which they are preempted and placed at the end of the queue.

Smaller time quantums trade CPU utilization for better response time.

lottery scheduling

A weighted random process is chosen to run during each time quantum.

The weight determines the probability of being chosen and ultimately the CPU time for that process.

Allows a process to distribute its CPU time across multiple child processes.

priority scheduling

Processes are assigned priority values, and the scheduler selects the process with the highest priority to run, either preemptive or non-preemptive.

Suffers from possible **starvation** and **priority inversion**, where a high priority process is blocked by a low priority process, e.g. due to a lock on a shared resource.

Decreasing priority over time can mitigate starvation.

Priority inheritance can be used to mitigate priority inversion, where a lower priority process that is locking a shared resource is temporarily given the priority of the higher priority process that is blocked on it until it unlocks it.

multi-level feedback queue (MLFQ)

MLFQ is a priority scheduling algorithm with multiple queues, where processes are moved between queues based on their CPU usage.

New jobs get the highest priority, and are moved to lower priority queues if they fully utilize their time quantum.

If a job gives up or blocks before fully utilizing its time quantum, it retains its priority.

Processes with equal priority are scheduled using RR.

4 Synchronization

The execution of concurrent processes may be non-deterministic due to the order of execution of instructions on shared, modifiable resources.

These **race conditions** or **data races** can lead to incorrect results due to unsynchronized access.

The code segment which accesses shared resources is known as a **critical section**, during which *at most one* process can execute at a time.

A proper critical section (CS) implementation must enforce the following:

- **mutual exclusion**: if a process is executing in the CS, all other processes must not enter the CS
- **progress**: if no process is in the CS, then one waiting process should be granted access
- **bounded wait**: after a process requests access to the CS, there must be an upper bound on the number of times other processes can enter the CS before it
- **independence**: a process not executing in the CS should never block other processes

Failure to enforce any of these properties can lead to the following:

- **incorrect program behaviour**: from lack of mutual exclusion
- **deadlocks**: when all processes are blocked due to a *lack of progress*
- **livelocks**: when deadlock avoidance mechanisms fail to make progress
- **starvation**: when a process is blocked forever

4.1 Peterson's Algorithm

Peterson's algorithm

A mutual exclusion algorithm for 2 processes.

- **`flag[2]`**: a boolean array of length 2
- **`turn`**: an integer

The `flag[]` array is used to indicate whether a process is requesting access to the CS.

The `turn` variable is used to indicate which process should be granted access to the CS.

```
/* PROCESS 0 */
flag[0] = true;
turn = 1;
while (flag[1] && turn == 1); // busy wait
/* critical section */
flag[0] = false;

/* PROCESS 1 */
flag[1] = true;
turn = 0;
while (flag[0] && turn == 0); // busy wait
/* critical section */
flag[1] = false;
```

Let the two processes be P_0 and P_1.

Peterson's algorithm achieves mutual exclusion as no state can satisfy both `turn == 0` and `turn == 1` simultaneously, so neither processes can be in their critical sections at the same time.

Note that while P_0 is in its CS, then `flag[0] == true`, and one of the following must hold:

1. `flag[1] == false`: P_1 just left its CS,
2. `turn == 0`: P_1 is waiting to enter its CS, or,
3. P_1 has yet to (but is about to) set `turn = 0`

Peterson's algorithm achieves progress as a process cannot immediately re-enter the CS if the other process has set its `flag` to `true`, which indicates that it wants to enter its CS.

Peterson's algorithm achieves bounded wait as a process will never wait for more than turn to enter its CS.

However, the disadvantages of Peterson's algorithm are its busy waiting (instead of blocking), its lack of general synchronization mechanism beyond mutual exclusion, and it being too low-level.

4.2 Test-and-Set

```
void enterCS(int* lock) {
   // testAndSet(*ptr) is an assembly level
   // instruction that is atomic:
   // *ptr = 1;
   // return old value of *ptr;
   while (testAndSet(lock) == 1);
}
void exitCS(int* lock) {
   *lock = 0;
}
// usage:
int lock = 0;
enterCS(&lock);
/* critical section */
exitCS(&lock);
```

Disadvantages of test-and-set are its busy waiting and lack of bounded-wait guarantee unless the scheduling is fair, leading to possible starvation.

4.3 Semaphores and Mutexes

A semaphore is a synchronization primitive which records the number of new processes that can safely access a shared resource, based on the number of processes that are currently accessing it.

A **general/counting semaphore** allows any integer number of these processes, while a **binary semaphore** only allows either 0 or 1.

```
int S = /* <initial value> */;
void wait() { // aka. P(), Down()
   // block while S <= 0
   // decrement S
}
void signal() { // aka. V(), Up()
   // increment S
   // unblock a waiting process, if any
   // this never blocks
}
```

Therefore, for some initial value of S, the current value of S is the initial value, plus the number of `signal()` calls, minus the number of `wait()` calls.

mutexes

Mutexes are implemented using a binary semaphore, to enforce mutual exclusion, but can run into deadlocks if used incorrectly.

```
wait(mutex);
/* critical section */
signal(mutex);
```

general synchronization

Sempahores can also be used to enforce that a section of P_1 must be executed only after a section of P_0.

```
/* PROCESS 0 */
produce(&X);
signal(semaphore);

/* PROCESS 1 */
wait(semaphore);
consume(X);
```

4.4 Threads

A single **multithreaded process** can have multiple **threads** each with their own ID, registers, and stack.

Threads share the memory context (only the text, data, heap), and OS context (PID, file descriptors, etc.), and as such are much lighter than processes, requiring much less resources (**economical and resource-sharing**).

Multithreaded programs are also more **responsive** with UI decoupled from program logic and **scalable** to multiple CPU cores.

user vs kernel threads

Threads can be either **user threads** or **kernel threads**.

The kernel is only aware of kernel threads, which it can schedule at the thread-level for better multi-core CPU utilization.

However, this comes with the overhead of kernel thread operations being system calls and less flexible than user threads which are implemented in libraries.

The effect of system calls on threads is OS-dependent. In Unix,

- **fork()**: process is duplicated, but the child process will only have one thread
- **exit()**: terminates the entire process, and all threads are terminated
- **exec()**: only the calling thread is replaced by the new program image

Part II

File Systems

Files can be viewed as a collection of logical blocks. When the file size is not a multiple of the block size, the last block may have **internal fragmentation**.

File systems keep track of these blocks, allow efficient access, and ensure effective utilization of disk space.

5 File System Implementation

Each partition on a disk contains 5 sectors:

1. **OS boot block**:
2. **partition details**: stores free space information
3. **directory structure**: stores directory information
4. **file info**: stores file allocation information
5. **file data**

5.1 Allocation

In **contiguous allocation**, blocks are stored in a contiguous 1D array with metadata on the starting indices and length (in blocks) of each file.

Advantage: efficient sequential and random access as offsets into a file can be used.

Disadvantage: starting location of each file is assigned randomly, leading to external fragementation (empty blocks between files).

In **linked list allocation**, each block in a file points to the next block in the file, and the addresses of the starting and ending blocks are recorded.

Advantage: no external fragmentation.

Disadvantage: pointer overhead per block, possible pointer corruption, sequential and random access is inefficient as the links must be traversed to reach the desired block.

file allocation table (FAT)

In a file allocation table, each block has an entry and is paired with the index of the next block in the file.

The FAT can be stored in RAM which makes random access efficient, but the FAT must be updated when a file is modified.

Furthermore, the FAT must record all disk blocks in a prtition, which on large disks can be memory space-inefficient.

In **indexed allocation**, the index of every block in a file is also stored in a block, such that only the index block needs to be read to access the file.

Advantage: less memory overhead, only index block is kept in memory, no external fragmentation.

Disadvantage: index block overhead, maximum file size is limited by the number of blocks in the index block.

Variations of the indexed allocation strategy include **linked indexes** and **multi-level indexing**.

5.2 Free Space Management

Using a **bitmap**, each block is marked as either free or allocated with `0` or `1` (implementation-dependent).

Advantage: bitwise manipulation is efficient.

Disadvantage: memory overhead as the bitmap must be stored in memory for efficiency.

Using a **linked list**, free blocks are linked together and this information is stored in a block. Free space can be allocated in reverse order of the linked list.

Advantage: only the first pointer needs to be stored in memory, and free blocks are easy to locate.

Disadvantage: high overhead if not storing the free block list in a free block.

5.3 Directory Structure

Directories are used to group files together, and are actually files themselves. They contain information of the files and sub-directories in the directory.

In a **linear list** implementation, each file and its starting index and length is stored in a list.

Disadvantage: linear search to find a file is inefficient, caching can offset this

In a **hash table** implementation, file names are hashed to a bucket, and their starting index and length are stored in a list in that bucket.

Advantage: fast lookup.

Disadvantage: hash collisions, chained collision resolution will lead to linear search over the length of the chain.

Another approach to storing file information is to use pointers to the metadata instead of storing the metadata directly.

5.4 Microsoft FAT File System

Each partition on a disk contains 5 sectors:

1. **boot block**
2. **FAT**: stores partition details
3. **duplicate FAT**: backup in case of FAT corruption
4. **root directory**: stores all file and directory information (FAT16 only)
5. **data blocks**: stores directory structure, file info and file data

Every entry in the FAT contains one of the following:

1. block number of the next block
2. `EOF` code (e.g. `NULL` pointer)
3. `FREE` code (unused block)
4. `BAD` code (unusable block, e.g. disk error)

For FAT16, there are up to 2^{16} blocks and each of the 2^{16} FAT entries is 16 bits.

For FAT32, there are up to 2^{32} blocks and each of the 2^{32} FAT entries is 32 bits.

directory entries

The root directory has its own sector in the partition, and other directories are stored in the data blocks of the partition.

Each file or subdirectory is stored as a **directory entry** and consists of:

1. file name, 8 bytes
2. file extension, 3 bytes
3. file attributes, 1 byte
4. reserved bytes, 10 bytes
5. creation date and time, 4 bytes
6. first disk block number, 2 bytes
7. file size in bytes, 4 bytes

File attributes are used to indicate whether a file is a file, directory, or special (e.g. read only, hidden, etc.).

The FAT version can be determined by the size of the first disk block number — 2 bytes for FAT16 and 4 bytes for FAT32, hence the reserved space.

Since the file name is limited 8 characters, in FAT32, longer file names are stored in separate directory entries. A tilde is used to denote that the file name continues into the next directory entry.

reading a file in FAT

Given the root directory, data blocks and FAT sectors, find the sub-directory within the root directory, and then find the directory entry for the file within the sub-directory.

Alternate between the data block and FAT table starting from the first disk block number until the `EOF` code is reached.

5.5 Extended-2 (Ext2) File System

Each partition is divided into the boot sector followed by multiple **block groups**.

Each block group contains:

1. **superblock**:
   metadata for the file system, e.g. how many block groups, how many I-nodes per group, etc.
2. **group descriptors**:
   metadata on where to find the following items, and duplicates all other group descriptors for redundancy
3. **block bitmap**:
   one bit for every data block, `1` for allocated and `0` for free blocks
4. **I-node bitmap**:
   one bit for every I-node, `1` for allocated and `0` for free I-nodes
5. **I-node table**:
   one 128-bytes entry for every file and directory, contains file information
6. **data blocks**:
   contain file data and directory structure

I-nodes (index-nodes)

Each I-node has the following structure:

1. mode (file type, permission bits, etc.), 2 bytes
2. owner information, 4 bytes
3. file size in bytes, 4 or 8 bytes
4. timestamps, 3 x 4 bytes
5. **data block pointers**, 15 x 4 bytes
6. **reference count**, 2 bytes

I-nodes use the combined allocation strategy with linked indices and multi-level indexing.

The first 12 pointers of the data block pointers point to **direct blocks** of 1 KiB each storing actual data.

The 13th pointer points to a **single indirect block**, which uses single-level indexing to point to the next 256 1KiB blocks.

(256 entries as the indirect block is also 1 KiB, and each address is 4 bytes.)

The 14th pointer points to a **double indirect block**, which uses double-level indexing for a maximum of 64 MiB.

The 15th pointer points to a **triple indirect block**, which uses triple-level indexing for a maximum of 16 GiB.

The **reference count** is used to determine when to free the I-node and its data blocks.

With multiple levels of indexing, both small and huge files can be handled and accessed efficiently.

directory structure

Directories are stored as I-nodes, and the data blocks of the I-node point to the files and sub-directories, forming **directory entries**.

Directory entries are stored in a linked list and each has associated with it:

1. an I-node number for the file or sub-directory,
2. an offset to skip to the next directory entry,
3. a flag for the file type, ’F’ for file and ’D’ for directory,
4. the size of the file in bytes,
5. and a variable-length file name.

Each directory entry is variable length due to the variable-length file names.

The final directory entry points to an I-node number of `0`.

Creating hard links involves new directory entries pointing to the same I-node, increasing the reference count of that I-node.

Creating symlinks incurs the overhead of new I-nodes and data blocks, to store the path to the file (incurring re-traversal overhead), but does not increment the reference count.

reading a file in Ext2

Given the I-node number for the root directory and the I-node table:

1. Find the I-node for the root directory.
2. Find the disk block number for the directory entries for the root directory from the I-node.
3. Access the disk block and find the I-node number of the sub-directory/file.
4. Repeat steps 2 and 3 until the I-node number of the file is found.
5. Access the disk blocks in the I-node to read the file data.