Lecture 9 Pipelining ================================

1. **Performance Calculation Formulas**
   1. **Non-Pipelined**
      a) **Single-Cycle Datapath**

E.g. Stage	Time
F	5
D	3
E	3
Mem	5
WB	3

T_k

$CT_{seq} = 18$

Cycle time:

- $CT_{seq} = \sum_{k=1}^{N} T_k$
- T_k = Time for operation in stage k
- N = Number of stages

Total Execution Time for I instructions:

E.g. 78 × 18ns = 1482ns

- $T_{seq} = \text{Cycles} \times \text{Cycle Time} = I \times CT_{seq} = I \times \sum_{k=1}^{N} T_k$

      b) **Multi-Cycle Datapath**

$= I \times \sum_{i \in G_i} \frac{n_i}{I} \times C_i \times CT_{multi}$

$= \sum_{i \in G_i} (n_i \times C_i) \times CT_{multi} = \sum_{i \in G_i} n_i \times t$

\# of insts, # of clk cycles, cycle time $= max(T_k)$

Cycle time:

- $CT_{multi} = max(T_k)$
- $max(T_k)$ = longest stage duration among the N stages

Total Execution Time for I instructions:

- $T_{multi} = \text{Cycles} \times \text{Cycle Time} = I \times \text{Average CPI} \times CT_{multi}$
- Average CPI is needed because each instruction takes different number of cycles to finish

2. **Pipelined Processor**

- Total Time needed for I instructions:

E.g. $(78+5-1) \times (5+1)$ — Cycles, $max(T_k)$, pipeline delay

$$Time_{multi} = \text{Cycles} * CycleTime_{pipeline}$$
$$= (I + N - 1) * ((max(T_k) + T_d)$$

I inst.s; N Cycles; Longest stage duration among the N stages; overhead for pipelining E.g. pipeline register delay

3. **Ideal Speedup**

Assumptions for ideal case:

- Every stage takes the same amount of time: $\sum_{k=1}^{N} T_k = N \times T_k$
- No pipeline overhead $\rightarrow T_d = 0$
- Number of instructions I, is much larger than number of stages N

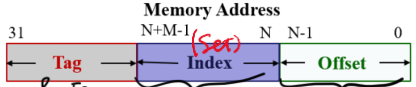
Note: The above also shows how pipeline processor loses performance

$$Speedup_{pipeline} = \frac{T_{seq}}{T_{pipeline}} = \frac{I \times \sum_{k=1}^{N} T_k}{(I+N-1) \times (max(T_k) + T_d)} = \frac{I \times N \times T_k}{(I+N-1) \times T_k} \approx \frac{I \times N \times T_k}{I \times T_k} \approx N$$

Reasons for best speedup < N: Imbalanced pipeline (some stages taking longer time), pipeline register delays

Lecture 10 Caches ==============================

1. **Memory Address Structure**



Cache Pattern for Eg.

Cache Block size = 2^N bytes — N

Number of cache blocks = 2^M / sets — M

Offset = **N bits**

Index = **M bits**

Tag = **32 – (N + M) bits**

2. **Calculating field lengths (n-way set associative)**

N = ① offset = $\log_2$(n. bytes/blk) = $\log_2$(4 × n. wd/blk)

M = index bit = ② $\log_2$(n. of sets) = $\log_2\left(\frac{\text{n. of blocks}}{\text{n-way}}\right)$

③ n. of tag bits = 32 − ① − ②

E.g. 2-way, total size = 16 words, 2 wds/blk (1 word = 4 bytes)

① offset = $\log_2(4\times2) = 3$

② index = $\log_2\left(\frac{16/2}{2}\right) = 2$

③ tag = 32 − 3 − 2 = 27

3. **Cache hit**

When is there a cache hit?

```
Valid[index] == True AND Tag[index] == Tag[memory address]
```

Lecture 11 Introduction to OS ======================

1. **Process Management**

1.1 Context Switching

When the CPU switches to another process, the system must **save the state of the old process** and **load the saved state for the new process** via a context switch. (CPT104)

Looks as if multiple processes are running simultaneously.

Steps:

a. Save the "context" (Register values) of the process to be suspended (Including general purpose registers, program counter and status register but excluding hardware configuration registers)

b. Restore the "context" of the process to be (re)started.

2. **Monolithic Kernels**

Kernels can be monolithic or microkernel.

2.1 Monolithic kernels:

a. **All major parts of the OS** – devices drivers, file systems, IPC, etc. running in "kernel space"

b. Bits and pieces of the kernel can be loaded and unloaded at runtime(using "modprobe" in Linux)

c. Popular examples of monolithic kernels: Linux, Windows

2.2 Microkernels

d. **Only the "main" part of the kernel** is in "kernel space"

  i. Which contains the important stuff like the scheduler, process management and memory management

e. The other parts of the kernel operate in "user space" as system services:

  i. The file systems, device drivers,, ...

  ii. Examples of microkernel OS: MacOS

2.3 Comparing and contrasting the two kinds of kernels:

f. Monolithic: Much faster (Switching modes costs a lot of time)

g. Microkernel: More secure

Lecture 12 Process Management ===============

1. **Program v.s. Process**

A **program** consists of (Not actually running): Machine instructions (and possibly source code),Data,A program exists as a file on the disk E.g. command.exe, winword.exe

A **process** consists of (can potentially run): Machine instructions (and possibly source code), Data, Context

- Exists as instructions and data in memory
- may be executing on the CPU

A single program can produce multiple processes.

- E.g. chrome.exe is a single program
- But every tab in Chrome is a new process.

2. **Execution Modes**

Programs usually run sequentially. Each instruction is executed one after the other. Having multiple cores or CPUs allows parallel (concurrent) execution. Streams of instructions with no dependencies are allowed to executed together.

A multitasking OS allows several programs to run concurrently

Interleaving, or "time slicing"

3. **Process and Process Management**

3.1 The Process Model

In single-core single processor:

At any one time, at most one process can execute

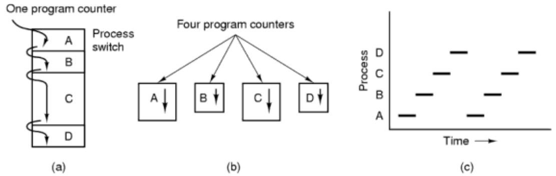
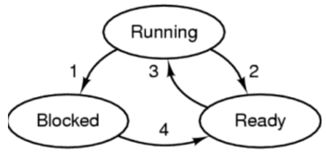


Figure (b) shows what "appears" to be happening in a single processor system running multiple processes:

Figure (c) illustrates how processes A to D share CPU time

3.2 Process States

3.3 Switching between processes



When a process runs, the CPU needs to maintain a lot of information about it. This is called the "process context"

a. **CPU register values**

b. **Stack pointers**

c. **CPU Status Word Register**

  i. This maintains information about whether the previous instruction resulted in an overflow or a "zero", whether interrupts are enabled

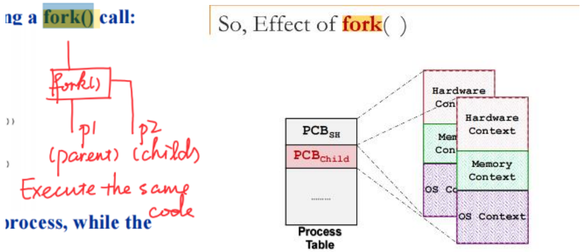
  ii. This is needed for branch instructions - assembly equivalents of if statements

The AVR Status Register – SREG – is defined as:

Bit	7	6	5	4	3	2	1
0x3F (0x5F)	I	T	H	S	V	N	Z
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Initial Value	0	0	0	0	0	0	0

4. **Process Creation**

A process can be created in Python by using a fork() call:

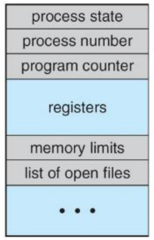


5. **Process Control Blocks**

a. When a process is created, the OS also creates a data structure to maintain information about that process:

  i. Called a "Process Control Block" (PCB) and contains:

  - Process ID (PID)
  - Stack Pointer
  - Open Files
  - Pending Signals
  - CPU usage



  ii. PCB is stored in a table called a "Process Table"

  - One Process Table for entire system
  - One PCB per process

b. When a process terminates:

(i) Most resources like open files, etc., can be released and returned to the system

(ii) However the PCB is retained in memory: Allows child processes to return results to the parent

(i) Parent retrieves the results using a "wait" function call, afterwhich the PCB is released

b. What if the parent never calls "wait"?

(i) PCB remains in memory

(ii) Child becomes a "zombie" process. Eventually process table will run out of space and no new process can be created

Lecture 13 Scheduling ============================

1. **Scheduling Environment**

1.1 CPU bound

Most of the time spent on processing on CPU

Graphics-intensive applications are considered to be "CPU bound"

Multi-tasking opportunities come from having to wait for processing result

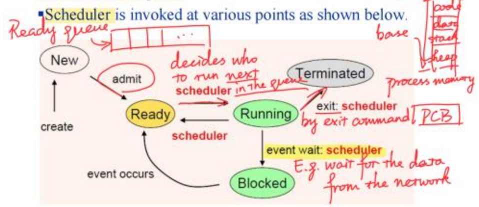
1.2 I/O bound

Most of the time spent on communicating with I/O devices

Multitasking opportunities come from having to wait for data from I/O devices

2. **Process States**

- **Processes switch between a fixed set of states depending on events that take place.**
  - Scheduler is invoked at various points as shown below.



3. **Types of Multitaskers + Scheduling Policies**

Policies are determined by the kind of multitasking environmentba

Simplest policy (Great for *all type* of multitaskers):**Fixed Priority**

3.1 Fixed Priority: Each task is assigned a priority by the programmer. Usually priority number 0 has the highest priority.

Tasks are queued according to priority number

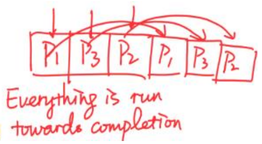
Batch, Co-operative. Task with highest priority is picked to be run next

Pre-emptive, Real-Time. When a higher priority task becomes ready, current task is immediately suspended and higher priority task is run

3.2 Batch Processing – First come first serve, Shortest Job First

Not actually multitasking since only one process runs at a time to completion

a. **First come first serve (Round-Robin scheduling)**



a) Arriving jobs are stored in a queue

b) Jobs are removed in turn and run

s.

c) Particularly suited for bath systems
d) Extension for interactive systems:
e) Jobs removed for running are put back into the back of the queue
f) Starvation free as long as earlier jobs are bounded

b. Shortest Job First

ential for starvation.

① p1: 200 transactions
② p2: 1000 ...
③ p3: 1200 ...
→ p4: 2000
→ p5: 100

Keep inserting short jobs ⇒ p2 may never be run

a) Processes are ordered by total CPU time used
b) Jobs that run for less time will run first
c) Reduces average waiting time if number of processes is fixed
d) **Potential for starvation**

3.3 Co-operative Multitasking – Round Robin with Voluntary Scheduling

Currently running processes cannot be suspended by the scheduler

Processes must volunteer to give up CPU time

3.4 Pre-emptive Multitasking – Round Robin with Timer, Shortest Remaining Time

a. Round Robin with Timer:

- Each process is given a fixed time slice *ci*
- After time *ci*, scheduler is invoked and next task is selected on a RR basis

b. Shortest Remaining Time

- Pre-emptive form of SJF
- Processes are ordered according to remaining CPU time left

Currently running processes can be force suspended by the scheduler

3.5 Real-Time Multitasking – Rate Monotonic Scheduling, Earliest Deadline First Scheduling

Processes have fixed deadlines that must be met

If don't meet the deadline:

a. Hard Real Time Systems: System fails
b. Soft Real Time Systems: Mostly just an inconvenience. Performance of system degraded.

4. Scheduling in Linux

Processes in Linux are dynamic: New processes can be created with **fork()**. Existing processes can exit.

Scheme ensures **no starvation of lowest priority processes.**

c. How process priorities are calculated:
   - Priority = base + f(nice) + g(cpu usage estimate)
   - f() = priority adjustment from nice value
   - g() = Decay function. **Processes that have already consumed a lot of CPU time are downgraded**

d. I/O boost:
   Rationale:
   Tasks doing read() has been waiting for a long time. May need quick response when ready
   Blocked/waiting processes have not run much
   Applies also to interactive processes - blocked on keyboard/mouse input

Lecture 14 Interprocess Communications =============

1. **Race Conditions:** Race condition occur when two or more processes attempt to access shared resources:Global variables, memory locations, hardware registers, CPU time
2. **Critical Section:** When a running process is reading/updating global variable (which can lead to race condition), it is within its "**critical section**"
3. **The Producer/Consumer Problem:** Consumer gets preempted, eventually no one is awake, deadlock
4. **Semaphores**

A semaphore is a special lock variable that counts the number of wake-ups saved for future use

A value of '0' indicates that no wake-ups have been saved

Two atomic operations on semaphore

a. DOWN, TAKE, PEND or P:

If the semaphore has a value > 0, it is decremented and the DOWN operation returns. **If the semaphore is 0, the DOWN operation blocks**

b. UP, POST, GIVE or V

If there are any processes blocking on a DOWN, one is selected and waken up. Otherwise UP increments the semaphore and returns.

4.1 Mutual Exclusion using Semaphore

When a **semaphore's counting ability is not needed**, we can use a simplified version called "mutex"

1 = Unlocked, 0 = Locked

Two processes can then attempt to DOWN the semaphore

Only one will succeed. The other will block

When the successful process exits the critical section, it does an UP to wake up others

4.2 Using Semaphores in Producer/Consumer Problem:

For two n-step (in assembly) process race conditions, the total possible cases are **n(n+1)/2**

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N; = 100
semaphore full = 0;
```

4.3 Deadlocks with Semaphores

Our producer/consumer solution swapped the semaphores for empty/full with the mutex semaphore, the potential deadlock occurs

5. Monitors and Conditional Variables

Monitors achieve mutual exclusion, but we also need other mechanisms for coordination. E.g. in our producer/consumer problem, **mutual exclusion is not enough to prevent the producer from proceeding when the buffer is full.**

We introduce "condition variable". One process WAITs on a condition variable and blocks, until... Another process SIGNALs on the same condition variable, unblocking the WAITing process

6. Barriers

A "barrier" is a special form of synchronisation mechanism that works with groups of processes rather than single processes. The idea of a barrier is that all processes must reach the barrier (signifying the end of one phase of computation)before any of them are allowed to proceed. **Can be implemented by semaphores.**

Lecture 15 Memory Management ===================

1. **Logical and Physical Addresses**

Logical addresses: These are addresses **as "seen" by executing processes code**

Physical addresses: These are addresses that are **actually sent to memory to retrieve data or instructions**

2. **Base and Limit Registers**

Base Register:

This contains the starting address for the program

All program address are computed relative to this register

Limit Register:

This contains the length of the memory segment

These registers solve both problems:

a. We can **resolve address conflicts by setting different values in the base register**
b. If a program tries to access memory **below the base register value** or **above the (base + limit) register value, a "segmentation fault" occurs**

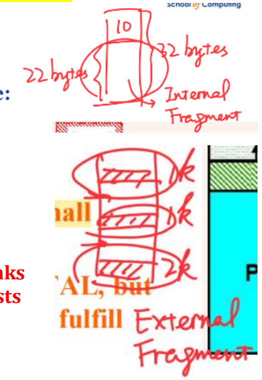
3. Partitioning issues: Fragmentation

Internal fragmentation:

Partition is **much larger than is needed**

Cannot be used by other processes

Extra space is wasted



External fragmentation:

Free memory is broken into small chunks by allocated memory

Sufficient free memory in TOTAL, but individual chunks insufficient to fulfils requests

4. Managing Free Memory

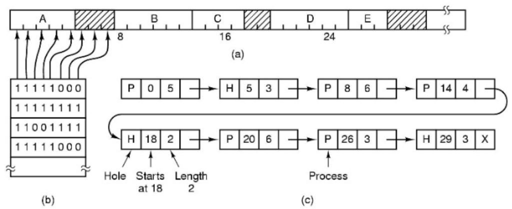
Two approaches:

a. Bit maps
b. Free/Allocated List

In either approach, **memory is divided up into fixed sized chunks called "allocation units"**

Common sizes range from several bytes (e.g. 16 bytes) to several kilo-bytes

Each "tick mark" in figure (a) represents the boundary of an allocation unit



4.1 bitmaps

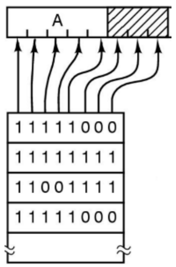
Each bit corresponds to an allocation unit

0 = free, 1 = allocated

Calculation questions:

① Max amount of memory managable = len(bitmap) in bits × $\underbrace{\text{n. bytes/bit}}_{\text{AU size}}$

1 bit ~ n bytes

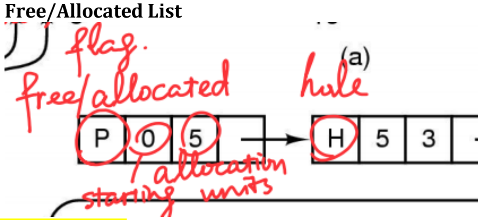


malloc: allocate in bytes

When allocating in bytes, **calculate their corresponding unit(s) in AU** (round to ceiling) and reflect in bitmap.

② Internal Fragmentation = Σ(n. of AU × size(AU) - actual num of bytes)

4.2 Free/Allocated List



5. Allocation Policies

5.1 First-Fit

a. Scan through the list/bit map and find the first block of free units that can fit the requested size
b. Fast, easy to implement

5.2 Best Fit

a. Scan through the list/bit map to find the smallest block of free units that can fit the requested size
b. Minimise waste
c. It can lead to scattered bits of tiny useless holes

5.3 Worst Fit

a. Find the largest block of free memory
b. Can reduce the number of tiny useless holes

$O(\log_2 n)$

5.4 Buddy Allocation

Binary splitting

Half of the block is allocated

The two halve are called "buddy blocks"

Can combine again when two buddy blocks are free

Lecture 17 File Systems ===========================

1. I-node in UNIX

Owner id

File type

Protection information

Mapping to physical disk

Time of creation, last use,

Reference counter

Inode calculation questions:

1. pointer size = $\frac{\text{len(blk idx) bits}}{\text{8 bit / byte}}$

n_p = Max # of pointers = $\frac{l}{\text{pointer size}}$

2. Max. size of a single file = $\sum n_{direct} \times l + \sum_{i=1}^{n} n_i \times n_p^i \times l$

where block size = l bytes

E.g. $\underbrace{10 \times 2048}_{\substack{\text{direct} \\ n_d = 10}} + \underbrace{512 \times 2048}_{\substack{\text{lvl 1} \\ n_1 = 1}} + \underbrace{512^2 \times 2048}_{\substack{\text{lvl 2} \\ n_2 = 1}} \cdots$

3. Max partition possible

$2^{\text{len(blk addr) bits}} \times l$

2. Linked List Allocation –

Linked list implemented a

4 types:

a. Block number (part
b. EOF byte (end of lin
c. FREE byte
d. BAD byte

Some numbers are block

block numbers for EOF, F

Blocks for file linked toge

3->5->8

Free blocks indicated by FREE entry

Bad blocks (unusable blocks due to disk error) marked in FAT entry: BAD cluster value

2.1 FAT16

Total # of blocks = **2^16** (- some reserved blocks)

Max size = Total # of blks * size (bytes/block)

2.2 FAT32

FAT32 2^{28} clusters

- Increase FAT size to 28 bits, cluster numbers 28-bits