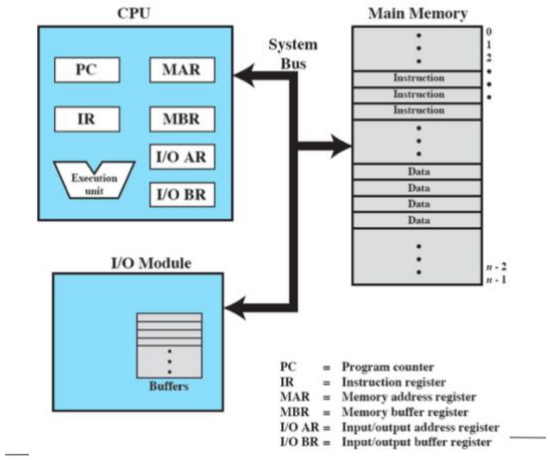
Table of Contents

Week 1

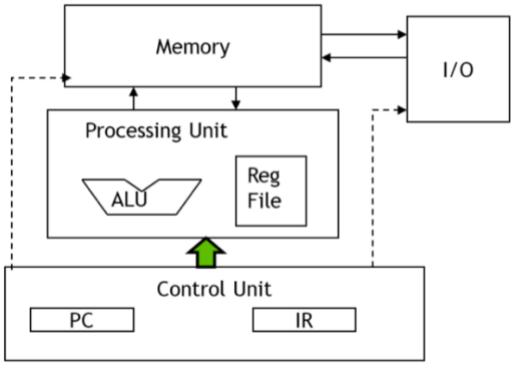
What is Computer system organization

(4 items)

- Processor
- I/O module
- System Bus
- main memory

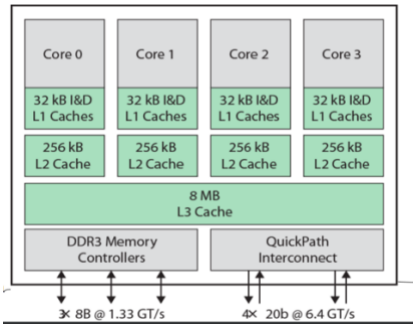


- At least 1 CPU.
- Device controller: disk controller, USB controller (to external hardware like mouse, keyboard, printer), graphics adapter (to monitor)
- All CPU & Device controller uses memory, and they compete for memory cycles.
- And these events are concurrent.



Multicore computer

- Each core is an individual processor
- Multicore chips also include L2 cache and sometimes L3 cache.



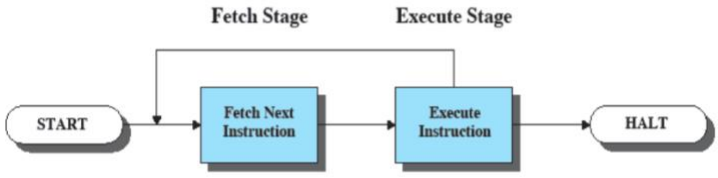
L1, L2, L3 cache

make the CPU more efficient, faster (L1>L2>L3) but more expensive than the main memory.

- Cache miss: when the CPU needs data but it is not in the L1/L2/L3, it needs to find it in the main memory, which is significantly slower.

How to execute Instruction

1. Processor reads instructions from memory
2. Processor executes instruction



Program counter holds **address of the instruction** to be fetched next, and **PC is incremented after each fetch.**

Interrupt

Processor pauses a running process, giving CPU resource to other processes, and resume it later to achieve **overall efficiency.**
For example, the CPU should not get blocked by a process waiting for I/O device delay.

Week 1 Homework

1. List and briefly define the four main elements of a computer
2. What is the purpose of interrupts?
3. What characteristics distinguish the various elements a memory hierarchy?
4. What is the distinction between spatial locality and temporal locality?
5. What are the main differences between multiprogramming and time sharing?
6. How are system calls implemented?

Week 2 Process

OS must allocate and protect resources, interleave execution of multiple processes, enable processes to share information and enable synchronization among processes.

Q1-1: what is a process

- Process is:

1. an instance of a program running on a computer;
2. a unit of activity characterized by the execution of a sequence of instructions, a current state, and an associated set of system resources.

- Two essential elements with a process: **Program Code** and **a set of data** with that code. We store them in the PCB (process control block).

Q1-2: what does PCB do and what stores in PCB?

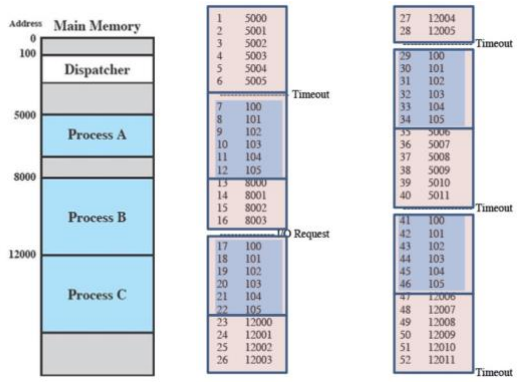
- PCB stores:
  - Program Identifier: Pid, to distinguish it from other processes.
  - State: RUNNING/SUSPENDED/READY…
  - Priority: priority level, 0, 1, 2..
  - Memory Pointers: the pointer to the program code and data associated with this process, plus any memory blocks associated with other processes.
  - Context data: data present in the register in the processor while its executing.
  - I/O status: includes the I/O requests, devices assigned to this process, list of files used by the process etc.
  - Accounting information: amount of processor time and clock time used, time limits, account numbers etc.
- PCB contains information so it's possible to **interrupt and resume execution**.
- Significance of PCB: enables OS to support multiple process and to provide for multiprocessing.

Q2-1: What is a state:
Refer to the Process 2 states/5 states model.

Q2-2: what is a dispatcher

- The dispatcher arranges which process is (continued) to be executed by the processor.
- The dispatcher process also runs on the processor, yet other programs can't see its instructions.

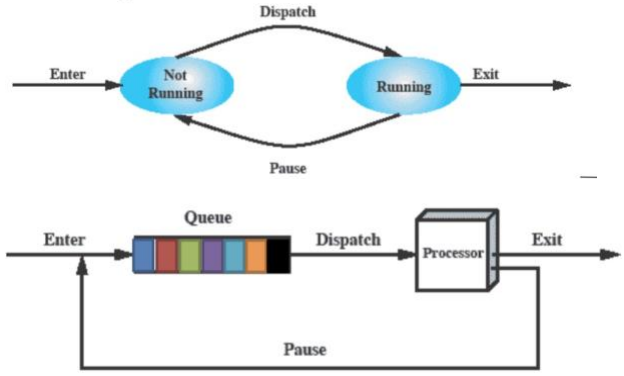
- The dispatcher process also lives in the main memory.



4 programs are running on a processor.
The blue areas are the dispatcher instruction code. (You can see it is repetitive)
Each process executes their own instructions in the correct order. (although being interrupted)

Q2-3: two state process model:

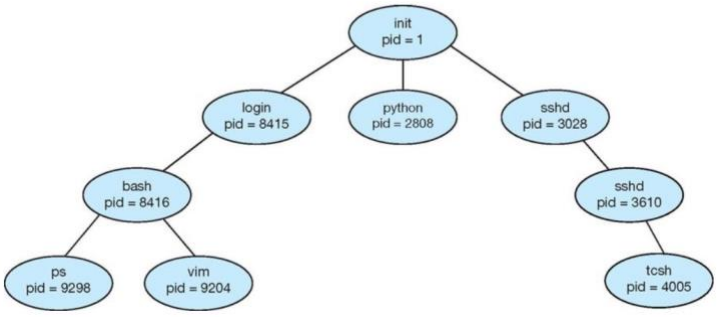
- Running /not running



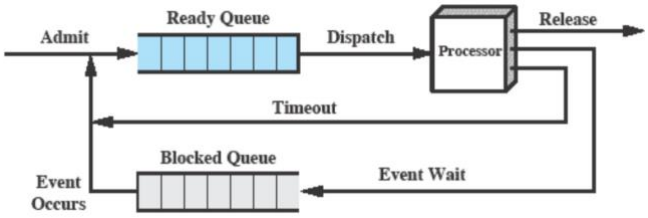
A process that is interrupted is transferred to the queue of waiting processes, alternatively, if the process has completed or aborted, it is discarded(exits). In either case, the dispatcher takes another process form the queue to execute.

Q2- 4: 5 state process model

- **Process Creation:** OS builds the data structures that are used to manage the process and allocates address space in main memory to the process.
- **4 reasons to create a new process**: new batch job, interactive log on (New user opening a terminal), created by OS for a service, and spawned by other process.
    - Details:
    1. Assign a unique PID to the new process
    2. Allocate space for the process
    3. Initialize PCB
    4. Set appropriate linkages
    5. Creates or expands other data structures.
    - **Process spawning:** a process created by other processes. **Parent process:** original, creating process. **Child process:** new process.



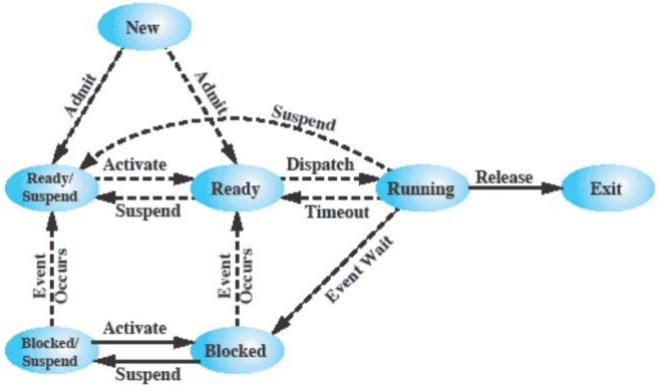
- **Process Termination:** 有很多种停止的原因, 包括 normal completion (by halt instruction, user action, parent process terminating), time limit exceeded, memory unavailable, bounds violation /protection error, arithmetic error etc.



- **Running:** The process that is currently being executed.
- **Ready:** A process in the Ready queue, it is prepared to execute when dispatcher picks it.
- **Blocked/Waiting:** A process that cannot execute until some event occurs. (so it gets back to the RR queue)
- **New:** a process and its PCB are created, but not yet admitted into the Ready Queue, usually not loaded into the main memory yet.
- **Exit:** a process that has been released, either because it halted or aborted.

- **Null → New:** new process created.
- **New → ready:** OS move process from new state to Ready state when it is prepared to take on an additional process.
- **Ready → Running:** When it is time to select a process to run, OS chooses one of the processes in the ready state
- **Running → Exit:** currently running process is completed or aborted
- **Running Ready:** current running process reached maximum allowable time (time quantum) or pre-emption occurs.
- **Running → Blocked:** A process is put in the blocked state if it requests something for which it must wait. (usually sys call or waiting OS to complete)
- **Blocked → Ready:** a process in blocked state back to ready when the event its waiting has happened.
- **Ready → Exit:** a parent may terminate a child at any time.
- **Blocked → Exit:** the comments under the preceding item apply

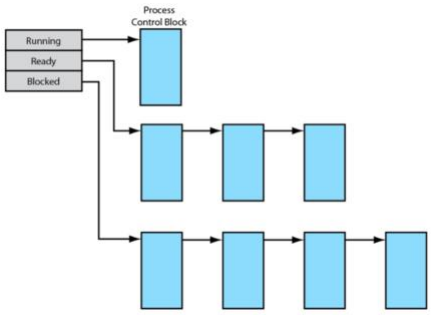
Q2-5 Suspended Process



- **Need for swapping**: The main memory is too small to load in all the processes.
  Since all process in the queues needs to be loaded into the main memory, and it's possible they're all waiting for IO, so that processor is idle for most of the time.
- **Steps to switch process:**
  1. Save the context of the processor
  2. Update the PCB for current process
  3. Move process to an appropriate queue
  4. Select another process for execution
  5. Update PCB of the new process
  6. Update memory management data structure
  7. Restore the context of processor to selected process state.

Q3 – describe data structures used by OS to maintain process management.

- Each queue is a linkedlist



- There is only one PCB in the running state. Other states (Ready, suspended, blocked) has multiple PCB nodes.

Modes of execution

- User mode: less privileged, user programs typically execute in this mode.
- Kernel mode: more privileged, for kernel of the OS.
- Why using two modes? Necessary to protect the OS and key operating system tables, such as process control blocks, from interference by user programs. In the kernel mode, the software has complete control of the processor and all its instructions, registers and memory.

Process switching

- A running process is interrupted, and the OS assigns another process to the running state and turns control over to that process.
- What event triggers it/ When to switch:
  Possible events:

Interrupt	External to the execution of the current instruction	Reaction to asyc external event
Trap	Associated with the execution of the current instruction	Handling of an error or an exception condition
Supervisor call	Explicit request	Call to OS function

  Interrupt: system interrupts. Control is first transferred to an interrupt handler,
    - Clock interrupt: OS determines whether the current running process time >= quantum
    - I/O interrupt: if the I/O action constitutes an event for which one or more processes are waiting, OS moves all blocked processes into Ready Queue, also it needs to determine whether to pre-empt the current running one.
    - Memory fault: processor encounters a virtual memory address that's not in main memory.

  Trap: an error or exception condition generated within the currently running process.
    - OS determines if its fatal, may attempt resume it or move to exit state, then make a process switch.

  Supervisor call: use of a sys call may place user process in the blocked queue.

Q5 – security

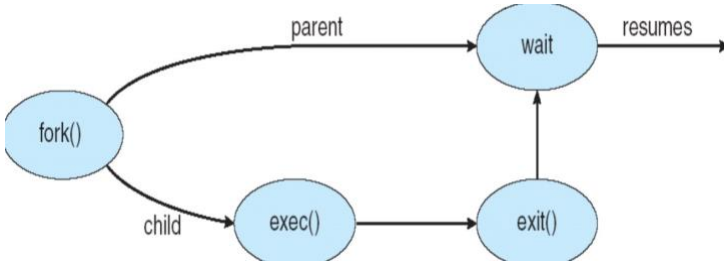
- System access threats: intruders/malicious software gets illegal access to root privilege
- OS should prevent any user/process to gain unauthorized privileges from the system – especially from gaining root access.

Q6 – Unix process creation & process control

- Fork() causes OS in kernel mode to do the following things:

1. Allocate a slot in the process table for the new process.
2. Assign a PID to child process
3. Copy of the process image of parent, with the exception of any shared memory
4. Increment the counters for any files owned by the parent to reflect an additional process now also own those files
5. Assign child process to Ready state
6. Returns ID of child to parent, and 0 to child.

Then kernel can either stay in parent, transfer control to child, or transfer control to another process.



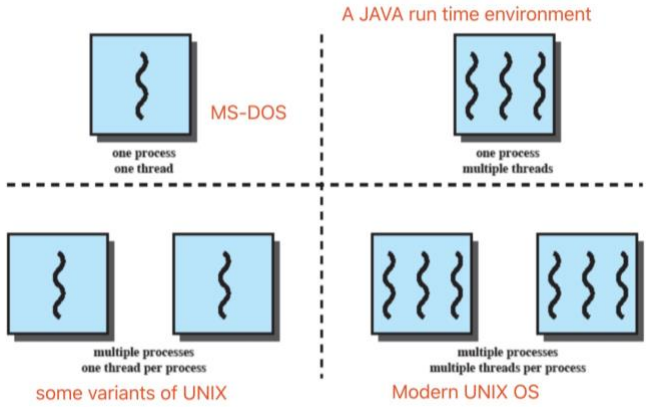
Week 2 – Threads

- Q1: Distinction between process and threads
- Q2: Describe basic design issues for threads
- Q3: Explain the difference between user level threads and kernel level threads

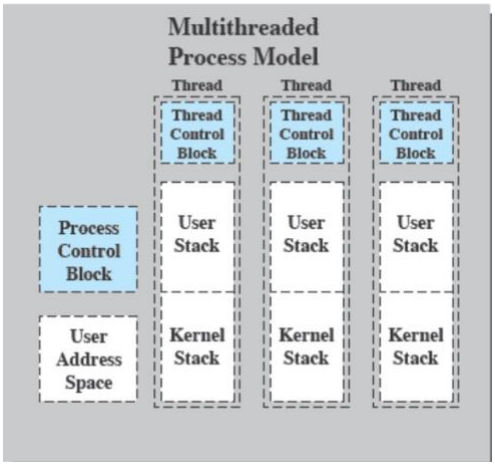
Two characteristics of processes:

- Resource ownership – process includes a virtual address space to hold the process image
- Scheduling/execution – follows an execution path that may be interleaved with other processes.
- Modern OS: dispatching → thread, resource ownership → process.

Multithreading: OS can support multiple, concurrent paths of execution within a single process.



- Process (in the Multithreading context):
  Unit of resource allocation and unit of protection. Associated with a virtual address space holding the process image; protected access to processors, other processes, files, I/O resources.
- Thread: all threads in the process can access the same memory and resources of its process, but does not share the execution state, saved thread context when not running, execution stack and thread static storage for local variables.

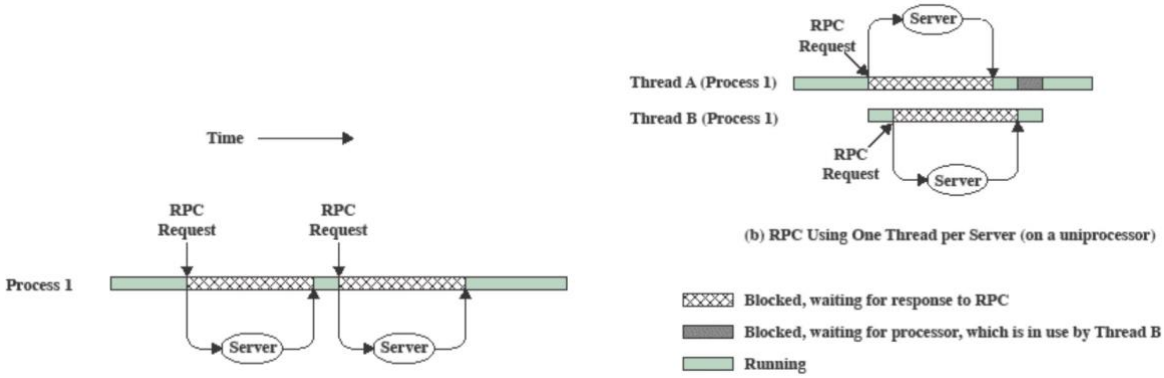


All of the threads of a process share the stat and resources of that process; they reside in the same address space and have access to the same data.
When one thread alters an item of data in the memory, other threads see the results when they access the item.
Multiple threads can read at the same time.

- Benefits of using threads:
  1. Takes far less time to create a new thread in existing process than creating a new process, less time to terminate a thread than a process, less time to switch between threads than processes

2. Threads enhances efficiency in communication between different executing programs. In most operating systems, communication between independent processes requires the intervention of the kernel to provide protection and the mechanism needed for communication.

- Benefits in an example:

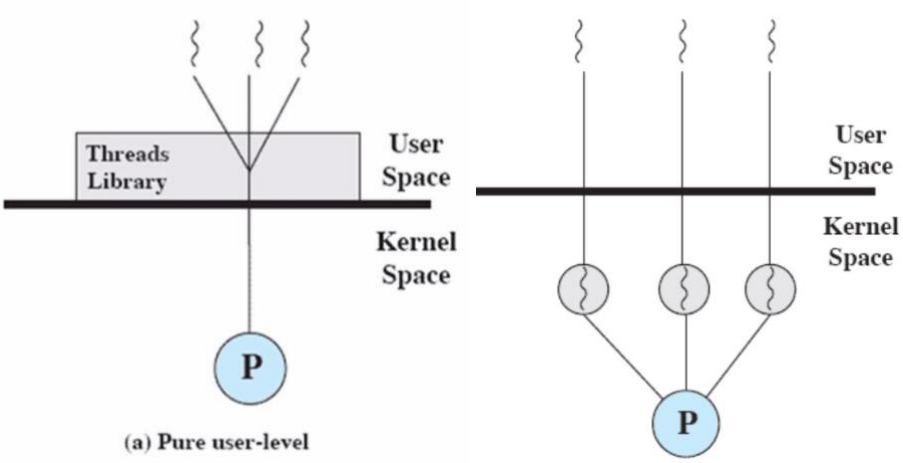


An example that multithreading speeds up significantly more than single thread. Notice here the 2 servers are different, not uniprocessing (otherwise has to be single thread)

Thread states:

- **Spawn:** when a process is spawned, a thread for that process is also spawned; or, a thread spawns another process.
- **Block:** when a thread waits for an event.
  – Note that thread is non-blocking.
- **Unblock:** when the event it's been waiting becomes available, thread goes back to RR.
- **Finish:** de-allocate register context and stacks.
- Why thread doesn't have suspended or terminate states? These states are process level. When a process is in suspend/terminate state, All of its threads are in the same state.

Q3 - Type of threads



- ULT: user level threads: all thread management is done by the application; kernel is not aware of the existence of threads, only schedule the process as a unit and assign a **single execution state** to that process. Spawned by user space threads library.
  - **Disadvantage of ULT:** Blocking, when a ULT executes a sys call, all of the threads in that process blocks. In pure ULT, a multi-threaded application cannot take advantage of multiprocessing; only a single user-level thread can execute at a time.
  - **Advantages of ULT:** thread switching does not require kernel mode privilege, because the thread management data is within user address space of a process; scheduling can be application specific – the scheduling policy can be independent of the OS; ULT can run on any OS.

- KLT: kernel level threads: light weight processes, non-blocking.
  Kernel maintains context information for the process and the threads; Scheduling is done on a thread basis;

    - **Advantage of KLT:** If one thread is blocked, the kernel can schedule another thread of the same process; kernel can simultaneously schedule multiple threads from the same process on multiple processors. Kernel routine themselves can be multithreaded.
    - **Disadvantages of KLT:** compare to ULT, KLT has the overhead of transferring control from one thread to another in the same process.
- **When to use KLT/ULT:** If most of the thread switch in an application require Kernel mode access, then ULT may not perform better than KLT.

Operation	User Level Threads	Kernel Level Threads	Processes
Null Fork	34	948	11300
Signal Wait	37	441	1840

Multicore & Multi-threading

- Performance improvement:

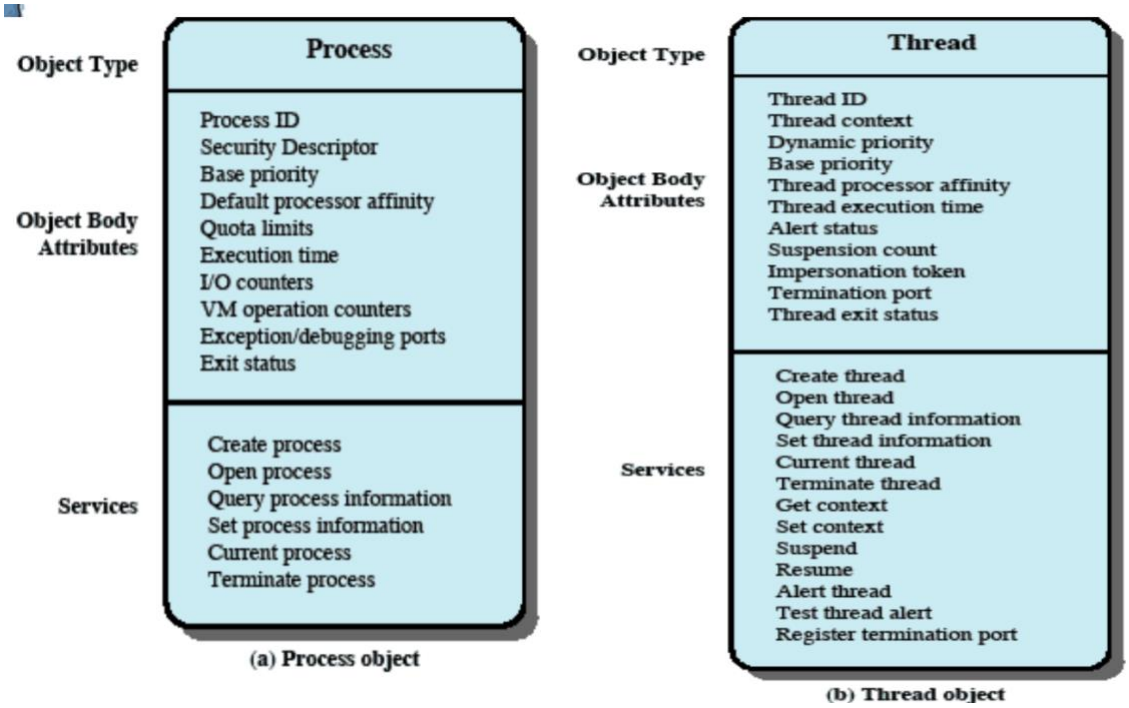
  $Speedup = \frac{time\ to\ execute\ program\ on\ a\ single\ processor}{Time\ to\ execute\ program\ on\ N\ parallel\ processors} = \frac{1}{(1-f)+\frac{f}{N}}$

  f = the proportion that is infinitely parallelizable.
  Even if f value is great, multiplying the N value won't result in N times acceleration.
  So it's not always better to increase the number of cores.
- **Soft affinity**: dispatcher tries to assign ready thread to the same processor it last ran on – this helps reuses memory cache.
  Hard affinity: dispatcher restricts thread execution on certain processors.
- **Disadvantage of Multicore:** putting more pressure on programmer to handle balancing, data dependency, splitting, dividing activities, testing & debugging.

Linux threads and Windows Threads

- **Linux threads** uses same internal representation for processes and threads; thread is simply a new process that happens to share the same address space as its parent;
- distinction is only made when a new thread created by the **CLONE** sys call. **FORK creates a new process with entirely new process context**; **CLONE** creates a new process with **own identity but allowed to share the data structures of its parent.**
- using clone gives an application fine grained control over exactly what is shared between two threads.

- Windows threads/processes are implemented as objects, created as new process or a copy of existing process; executable process may contain 1-N threads. Both processes and thread objects have built-in sync capabilities.
- Processes: entity corresponding to a user job or application that owns resources
- Threads: a dispatchable unit of work that executes sequentially and interruptible.



Week 3 Mutual exclusion and synchronization

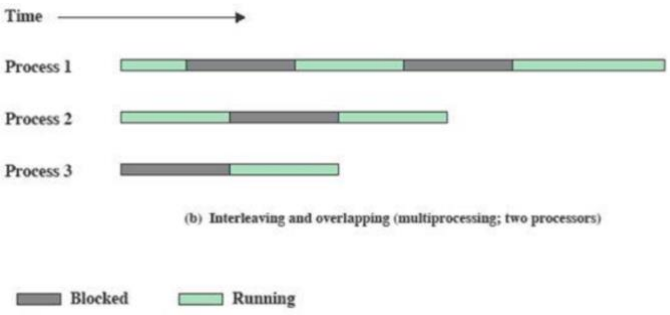
Modern OS allows

1. Multiprogramming
2. Multiprocessing
3. Distributed Processing

Big issue is concurrency: to manage the interaction of processes.

Discuss basic concepts related to concurrency, such as race condition, OS concerns, mutual exclusion requirements.

Processes can be overlapped on different processors



Resource Competition

- Concurrent processes come into conflict when they're competing for use of the same resource (I/O, memory, processor time..)
- Need to account for : deadlock, starvation, need for mutual exclusion.

Race condition

- When multiple processes and threads **read and write data** items so that the **final result depends on the order of execution of instructions** in the multiple processes.
- **The output depends on who finishes the race last. (depends on the loser)**
- For example, if two threads both add 1 to the same variable N, the result should be N+2. But thread two reads before thread one writes to the variable, so the result becomes N+1.

Critical Section

- When a process executes code that **manipulates shared data/resources, we say that the process is in its critical section**
- Need to design a protocol that process can use to corporate.

Mutual exclusion

- **Mutual exclusion** happens when several processes require access to **a single non-sharable resources**. Such resource is **critical resource** and the program that uses it is the **critical section.**
- Only one process at a time is allowed in the critical section for a resource; No assumptions are made about relative process speeds or number of processes.
- A process must not be delayed access to a critical section when there is no other process using it.
- A process halts in its non-critical section must do so without interfering other processes.
- A process holding the lock/resource must release the lock/resource under any circumstance – including throwing an error. That's why we usually put critical section in ***try/exception*** block, and releases the in the ***finally*** section.

Hardware support

- **Interrupt disabling**:
  Since process runs until it invokes an OS service or until it is interrupted, you can disable interrupts to guarantee mutual exclusion – works in uniprocessor systems.
  Disadvantage: does not work In multiprocessor architecture because it no longer guarantees mutual exclusion; lower the efficiency of execution.

- CAS(compare and swap): set a flag value and see if it's been changed.

```
/* program mutualexclusion */
const int n = /* number of processes */;
int bolt;
void P(int i)
{
   while (true) {
      while (compare_and_swap(bolt, 0, 1) == 1)
           /* do nothing */;
       /* critical section */;
       bolt = 0;
       /* remainder */;
   }
}
void main()
{
   bolt = 0;
   parbegin (P(1), P(2), ... ,P(n));
}
```

Busy waiting

```
int compare_and_swap (int *word,
   int testval, int newval)
{            =0          =1
   int oldval;
   oldval = *word;
   if (oldval == testval) *word = newval;
   return oldval;
}
```

- If `word = 1`, unchange, and return `1`
- If `word = 0`, `word = 1`, and return `0`

It has the down side of ***busy waiting***. (loop keep checking on a value)

- Exchange Instruction

```
   /* program mutualexclusion */
int const n = /* number of processes**/;
int bolt;
void P(int i)
{
   int keyi = 1;
   while (true) {
      do exchange (keyi, bolt)
      while (keyi != 0);
      /* critical section */;
      bolt = 0;
      /* remainder */;
   }
}
void main()
{
   bolt = 0;
   parbegin (P(1), P(2), ..., P(n));
}
```

Busy waiting

```
void exchange (int register, int
  memory)
{
   int temp;
   temp = memory;
   memory = register;
   register = temp;
}
```

Still has busy waiting issue.

Advantage of machine instruction

- Applicable to any number of processes on either a single processor or multiple processors sharing main memory. Simple and easy to verify.
- Can be used to support multiple critical sections; each critical section can be defined by its own variable.

Disadvantage of machine instruction

- Possible to have deadlock – If we have many priority, lower Priority process (lower P) wants to give lock to higher P , but the higher P wants to give lock to lower P due to mutual exclusion, resulting in deadlock.
- Busy waiting – wasting processor time checking over and over again, doing no actual job.
- **Starvation** is possible, when a process leaves a critical section, and **more than one process is waiting – the selection of next process to use is arbitrary.**

Week 3 – OS sync mechanisms

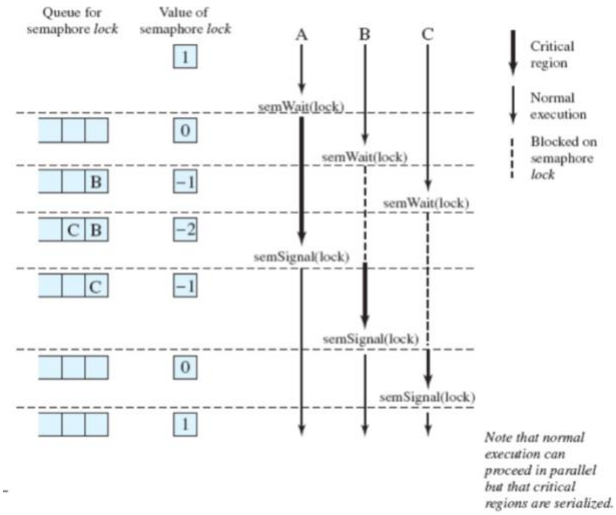
Semaphore – including mutex

```
struct semaphore {
    int count;
    queueType queue;
}
void semWait(semaphore s){
    s.count--;
    if (s.count < 0){
        /* place this process in s.queue */
        /* block this process */
    }
}
void semSignal(semaphore s){
    s.count++;
    if (s.count <= 0){
        /* remove a process P from s.queue */
        /* place process P on ready list */
    }
}
```

A queue is used to hold processes waiting on the semaphore – this **prevents busy waiting.**
An integer value is used for signalling among processes
Only three operations may be performed on a semaphore, **all of them are atomic** – initialize the integer, decrement (wait), increment. (signal)
If the integer gets < 0, it is blocked and wait. The negative number equals the number of blocked process waiting to be unblocked.
Each sem signal increments the integer, and unblocks one of the waiting processes when s is negative.
(looks like the producer/consumer model)

- Binary semaphore only allows the integer to be 0 or 1. When it is 1, it allows semWait, disables semSignal; when it is 0, it allows semSignal, disables semWait.

Strong semaphore: Sequence in process queue is FIFO, the one waited for the longest time gets the lock first
Weak semaphore: Sequence is not guaranteed.



- Producer consumer problem:
- 1 ~ N producer generates data and place them in a buffer
- 1~ N consumers takes item out of the buffer once at a time.
- Process time > 0.
- Only 1 producer/consumer can access the buffer at any one time.
- Need to ensure producer can't add data into full buffer and consumer can't remove data from empty buffer.

- Issues with semaphore:

1. semSignal() and semWait() are used among several processes, difficult to understand their effects
2. Usage must be correct in all the processes; one bad process can kill the whole system.
3. **Deadlock:** if semaphore is used wrong. Each thread must acquire resources using the same sequence. release 顺序无所谓.

P_0	P_1
semWait (S);	semWait (Q);
semWait (Q);	semWait (S);
…	…
semSignal (S);	semSignal (Q);
semSignal (Q);	semSignal (S);

4. **Starvation:** a process may be never removed from the queue. (如果是 weak 的话是乱出的)

Monitors

A programming language construct that provides equivalent functionality of semaphore but easier to control.

Messages

Useful for the enforcement of synchronization discipline.
Exchange information

Week 4 Deadlock and starvation

目标:
Q1: List and explain the conditions for deadlock
Q2: Define deadlock prevention and describe deadlock prevention strategies related to each conditions for deadlock
Q3: Explain the difference between deadlock prevention and deadlock avoidance
Q4: understand two approaches to deadlock avoidance
Q5: explain the fundamental difference in approach between deadlock detection and deadlock prevention
Q6: Understand how an integrated deadlock strategy can be designed
Q7: Concurrency and sync methods used in UNIX, LINUC, Win7

Deadlock: **permeant blocking** of a set of processes that complete for system resources or communicate with each other; these processes are deadlocked when each process in the set is blocked awaiting an event that can only be triggered by another blocked process in the set. No efficient solution to resolve an already occurred deadlock.

All deadlocks involve conflicting needs for **reusable resources** by **two/more processes**. What is reusable resource? Resources that can only be used by one process at a time and released by that process, like memory spaces. Deadlock may occur if each process holds one resource and request another.

Consumable resource: one that can be created and destroyed. Eg. Interrupts, signals, messages, information on I/O buffers.
A deadlock can also occur if "**receive**" is blocked; although it may take a rare combination of events to cause a deadlock (a program can be used for a considerable period of time before the deadlock occurs)

- Resource allocation graph

(a) Resouce is requested (b) Resource is held

If graph contains no cycles → no deadlock
If graph contains a cycle, only one instance per resource type → deadlock if no pre-emption
If graph contains a cycle, several instance per resource type → possibility of deadlock

Q1 - conditions for deadlock

Mutual Exclusion	Only one process may use a resource at a time
Hold-And-Wait	A process may hold allocated resources while awaiting assignment of others
No-pre-emption	No resource can be forcibly removed from a process holding it
Circular Wait	Closed chain of processes exists; each process holds at least one resource needed by the next process in chain

Q2 – Strategies to deal with deadlock

1. **Q3 - Prevent a deadlock:** adopt policy that eliminate one condition for deadlock
    a. **Mutual exclusion** must be hold for non-shareable resources, therefore cannot be relaxed.
    b. **Hold and wait** – guarantee whenever a process requests a resource it cannot hold any other resources(Either starts when you acquired all resources, or start waiting when you have 0 resources). Disadvantage: starvation
    c. **No pre-emption –** if process A that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently held by A are released; pre-empted resources are re-added to be added again. **Disadvantage**: not practical, live lock.
    d. **Circular wait –** impose total ordering of all resource types, require each process requests resources in an increasing order.
2. **Q4- Avoid deadlock**: make the appropriate dynamic choices based on the current state of resource allocation
    a. A decision is made dynamically whether the current resource allocation will potentially lead to deadlock.

b. Disadvantage: requires knowledge of future process requests
c. Restriction: Maximum resource requirement for each process must be stated in advance
d. Restriction: Process under consideration must be independent, with no sync requirement, and no process may exit while holding resources.

3. Detect deadlock: attempt to detect the presence of deadlock and take action to recover
    a. Adv: leads to early detection
    b. Disadv: consume processor time.

Dining philosopher

Q7 – Unix concurrency mechanism

- **Pipes:** circular buffer allowing 2+ process to communicate on producer – consumer model. FIFO queue. One process read, another write. Can be named/unnamed.
- **Messages:** a block of bytes with an accompanying type. **msgsnd/msgrcv** system calls for processes to engage in message passing. Associated with each process is a message queue.
- **Shared memory:** fastest inter-process communication. Common block of virtual memory shared by multiple processes. Permission is read-only / read-write for a process. **Mutual exclusion must be provided by participating process.**
- **Semaphores:** explained.
- **Signals:** delivered by updating a field in the process table for the process to which the signal being sent.

Q7 – Linux Kernel Concurrency Mechanism

- Barrier
- Spinlock:
  Most common in Linux; can only be acquired by one thread at a time. Any other tread will try spinning until it can acquire the lock; effective in situations where the waiting time for a lock is short; However the locked-out ones are in busy-waiting mode.
- Atomic operation: executes without interruption & without interference, including integer operation and bitmap operation.

Week 5 – Uni-Processor Scheduling

Resource: Processor provides execution time. Need a schedule to best allocate the jobs.

Aim:

assign process to be executed by the processor in a way that meets system objectives

- Share time fairly among processes
- Prioritise processes when necessary
- Prevent starvation of a process
- Use processor efficiently
- Low overhead.

Long term scheduling

Aim: determines which programs to admit into system.

- Controls the degree of multiprogramming. If admitting more processes, there is smaller percentage of time each process is executed.
- Can be FIFO, or consider some priority, execution time, I/O requirement.

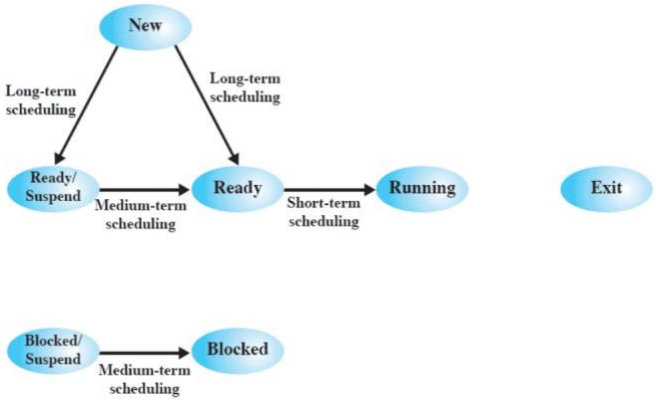
Medium term scheduling

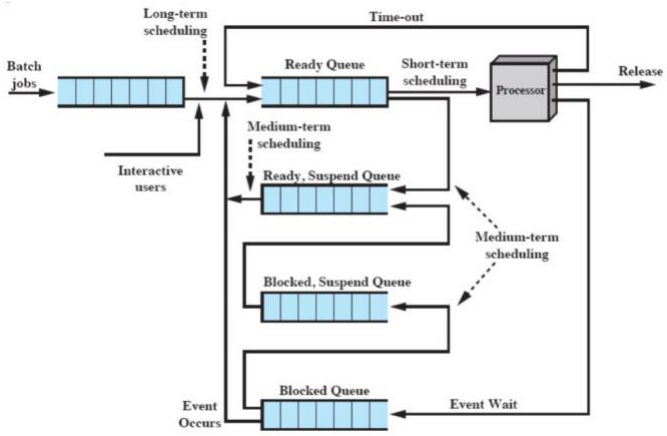
Aim: Swap in. Part of the swapping function.

Swapping-in decisions are based on the need to manage the degree of multiprogramming. Considers the memory requirements of the swapped-out process.

Short-Term scheduling

Aim: dispatcher. Allocate processor time to optimize system behaviour. Determines who to execute first.





Scheduling criteria

- User-oriented criteria

Relate to the behaviour (response time) of system perceived by an individual user/process.
Important on all systems. For a user's point of view, response time is always the most important.

- System oriented criteria

Focus on effective and efficient utilization of the processor. Less important on single user systems.

- Performance related: quantitative, easily measured.
- Non-performance related: qualitative, hard to measure. E.g. Predictability.

Scheduling Policy Terms

- Non-pre-emptive: once a process is in running state, continue until terminates or blocks.
- Pre-emptive: currently running process may be interrupted and moved to the ready state by the OS.
- Priority: scheduler will always choose a process of higher priority over one of lower priority. Multiple ready queues to represent each level of Priority.
- Starvation: lower priority may suffer starvation if there is a steady supply of higher priority processes. Solution: dynamic priority.

First Come First Served (FCFS)

- Each process join the ready queue in an FIFO manner.
- When the current process ceases to execute, the process waiting the longest time in the ready queue is selected.
- Disadvantage: a short process may have to wait a very long time before it can execute. I/O bound process have to wait until CPU bound ones completes.
- Advantage: easy, low overhead, favours CPU-bound process.

Round Robin – pre-emption based on a clock.

- Clock interrupt is generated at periodic intervals
- When an interrupt occurs, the currently running process is placed in the Ready Queue. The next ready job is selected.
- Good to select a time quantum slightly greater or equal to a typical job.

Shortest job first (SJF/SPN)

- Non pre-emptive.
- Process with the shortest expected processing time is selected next.
- Shortest process jumps ahead of longer processes.
- **Disadvantage:** predictability of longer processes is reduced; possibility of starvation for longer processes; if estimated time for process not correct, the OS may abort it.

Shortest Remaining time

- Pre-emptive version of Shortest Job first.

- Scheduler always chooses the process that has the shortest expected remaining processing time.
- Disadvantage: must be able to estimate processing time; risk of starvation for longer processes
- Advantage: better turnaround time compare to SPN because a short job is given immediate preference to a running longer job.

Feedback scheduling

- Dynamic priority
- Don't know the remaining time process needs to execute
- Penalize jobs that runs longer.
- If using a fixed quantum == Round Robin. Usually uses q = 2^i to improve performance

Feature and problem of Priority Scheduling

- Explicit priority: a priority number is associated with each process.
- Implicit priority: SJF is a priority scheduling where priority is the predicted CPU time.
- Problem: starvation

Highest Response Ratio Next

- Choose the next process with the greatest $Ratio = \frac{time\ spent\ waiting + expected\ service\ time}{expected\ service\ time}$

Priority Inversion – higher priority waits for lower priority

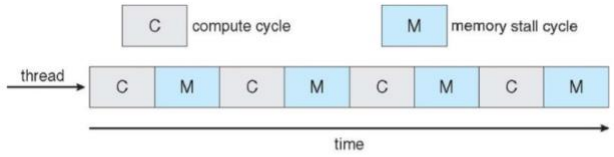
- OS force a higher priority task to wait for a lower priority task
- Can occur in any priority based pre-emptive scheduling scheme, Particularly relevant in the context of real-time scheduling.
- Unbounded Priority Inversion: the duration of a priority inversion depends not only on the time required to handle a shared resource, but also on the unpredictable actions of other unrelated tasks.
- A lower priority task should inherit the priority of any higher priority task waiting on a resource that they share.

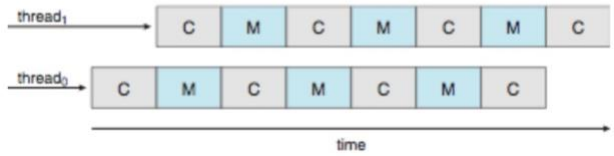
Contemporary Scheduling - pre-emptive.

- CPU scheduling uses time quantum determined by interval timer.
- Priority based process selection: select the highest priority process; priority reflects policy.
- Usually a variant of Multilevel Feedback Queues.

Thread Scheduling

- When OS supports threads, then scheduling is based on thread not process.
- There's a distinction between user level and kernel level threads
- SMP (Symmetric multiprocessing): Scheduling is more complex when multiple CPUs are available. Each Processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes.
  SMP system needs multicore processor. allow several threads to run concurrently. Each core maintains its architectural state and appear to the OS as a separate CPU. SMP system are faster and consume less power than systems where each processor has its own physical chip.
- Memory Stall: When a processor access memory, it needs significant amount of time waiting for the data to become available. Occurs because modern processors operate much faster than memory.





- How to load balance in SMP?
  If SMP, need to keep all CPUs loaded for efficiency. Load balancing attempts to keep workload evenly distributed.
  Push migration: periodic task checks load on each processor, if found task from overloaded CPU, push to other CPU.
  Pull migration: idle processors pulls waiting task from busy processor.

Algorithm Evaluation

- Deterministic modelling
  Analytical evaluation takes a particular predetermined workload and defines the performance of each algorithm for that workload. For each algorithm, calculated min average waiting time.
  ProCon: simple and fast, but requires exact numbers for input and the outcome is not general.
- Queuing Models
  - Describes the arrival of process, CPU, I/O bursts probabilistically. Computer system described as network of servers. Each with a queue of waiting processes. Knowing arrival rates and service rates, computes utilization, average queue length, average waiting time etc.
  - Cons: accuracy is not guaranteed.
  - Little's formula: $n = \lambda W$. $\lambda = average\ arrival\ rate\ into\ queue; W = average\ waiting\ time\ in\ queue; n = average\ queue\ length$
- Simulations:
  - More accurate, programmed model of computer system.
  - Data to drive simulation generated via random number generator according to probabilities.
  - You can trace tapes record sequence of real events in real systems.

Week 6 Memory management

Purpose

Memory Management requirements:

- Relocation, Protection, Sharing, Logical organization, Physical organization.

Relocation

Available main memory is shared among a number of processes. We need to swap in active process in and replace inactive processes. A process may be swapped to disk and returned at a different location. Memory address must be translated to the actual physical memory address.

Protection

Each process should be protected against unwanted interference by other processes, whether accidental or intentional. Without permission, other processes should not be able to reference memory locations in a process. Location of a program in main memory is unpredictable; memory references generated by a process must be checked at run time.

Sharing

protection mechanism must have the flexibility to allow several processes to access the same portion of main memory – some programs might want to use the same copy of a program.

Must allow controlled access to shared areas without compromising protection

Logical Organization

Main memory and Secondary memory are both organized into a linear / One dimensional address space. However, most programs are organized into modules (some are modifiable, some aren't)

The benefit of OS being able to deal with user programs/data in form of modules includes:

1. Modules can be written and compiled independently.
2. Modest additional overhead & different degree of protection.
3. Possible to share modules among processes.

Physical Organization

Main memory: fast access, relatively high cost, volatile (no permanent storage)

Secondary memory: large capacity, long term storage.

Contiguous Memory partitioning:

- OS divides memory up into relatively large partitions
- One process per partition.

Paging and segmentation:

- Non-contiguous memory allocation;
- OS divides memory up to relatively small blocks and processes are loaded in smaller pieces.

Continuous Partitioning – Fixed Partitioning

Cons in all fixed partitioning method variations: **number of partitions specified at system generation time limits the number of active process in the system**. Small jobs will not utilize partition space efficiently.

Equal size partitions

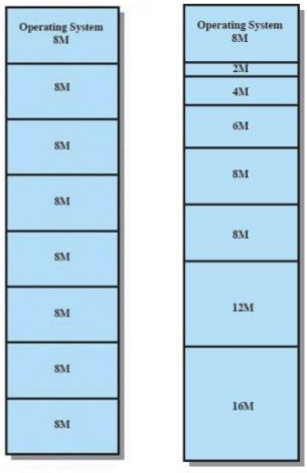
any process whose size is less than or equal to the partition size can be loaded into an available partition.

Cons:

- Inefficient memory utilization.
- **Internal fragmentation**: wasted space due to the block of data loaded being smaller than the partition.

Unequal size partitions

Similar to equal size partition but using a different partitioning division.



Continuous Partitioning – Dynamic Partitioning

Partitions are of variable length and number, process is allocated exactly as much memory as it requires.

Disadvantages in all variations

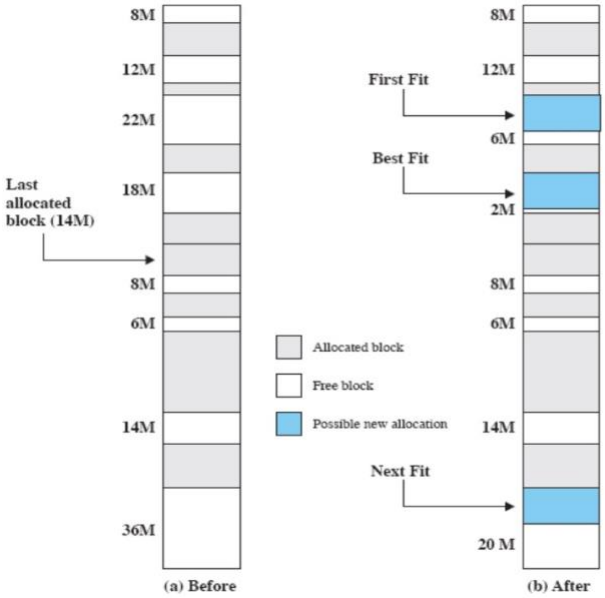
- External fragmentation: memory external to process is fragmented (enhances over time)
- The solution is Compaction: OS move process so that they are contiguous – free memory becomes together in one block. However, this is time consuming and wastes CPU time.

Dynamic Partitioning: dynamically allocate memory to processes.

The result of DP: memory becomes more and more fragmented. Utilization declines. Compaction can group some split fragments together. Moves processes so that they're contiguous – free memory that are together in one block.
But DP is time consuming & wastes CPU time.

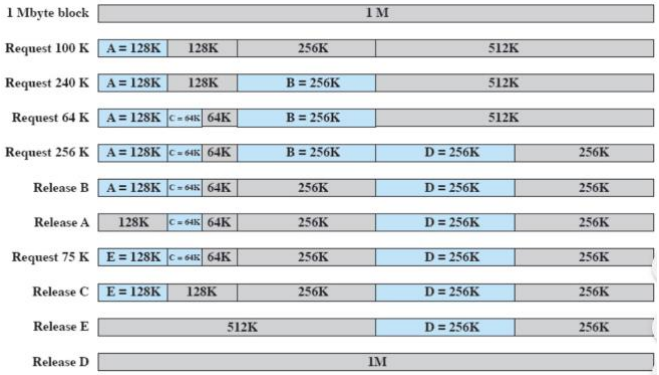
Best fit, first fit, next fit

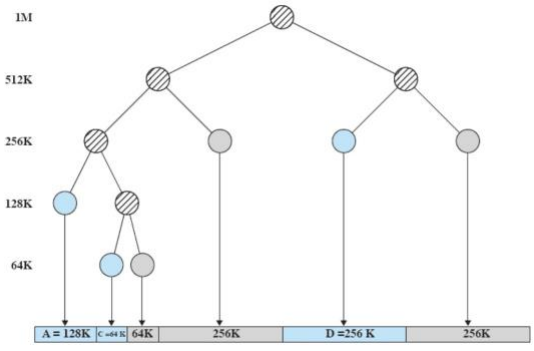
- **Best fit:** choose the block that is closet in the size to the request.
- **First fit:** begins to scan memory from the beginning, choose the first one large enough
- **Next fit**: begin to scan memory from the location of last placement, similar to First fit.



Continuous Partitioning – buddy system

Uses both fixed and dynamic scheme.
Memory blocks are available of size 2^k, where $L \leq K \leq U$, 2^L is the smallest block, 2^U is the entire memory space, load memory into the smallest fitting one.

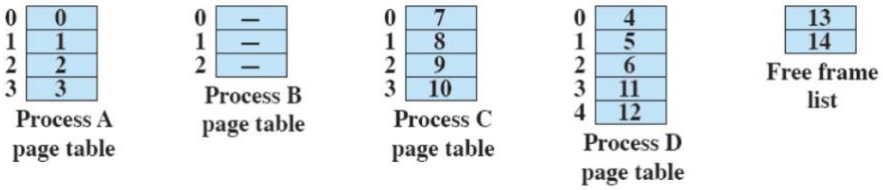




Non-Continuous Partitioning – Paging

Partition memory into equal fixed-size chunks that are relatively small. Process is also divided into small fixed size chucks of the same size.
Pages: chunk of a process
Frame: chunk of memory.

Page table: maintained by operating system for each process. Contains the frame location for each page in the process. Processor must know how to access for the current process. Used by processor to produce a physical address.

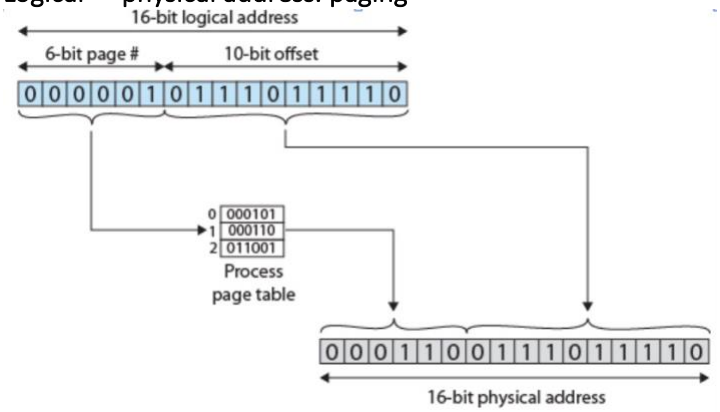


Segmentation - helps eliminate internal fragmentation.

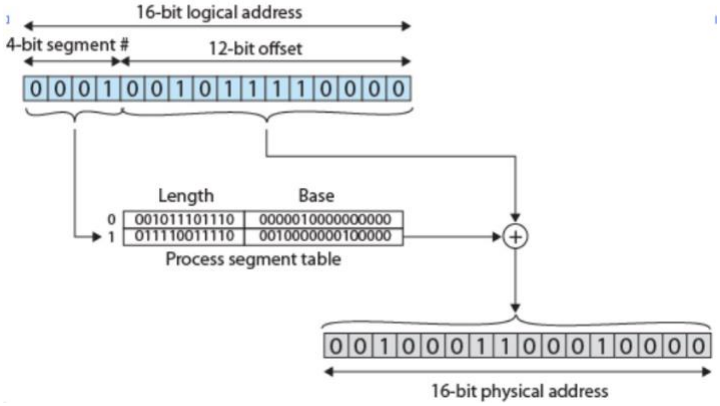
A program can be subdivided into segments, may vary in length (with max cap).
addressing consist of two parts:

- segment number
- offset.

Logical → physical address: paging



Logical → physical address: process segmentation table.



Security issue in memory management

- If a process has not declared a portion of its memory to be sharable, no other process should have access to the contents of that portion of memory.
- If a process declares that a portion of memory may be shared by other designated processes, then the security service of the OS must ensure that only the designated processes have access.

Buffer-overflow Attack

- Occurs when a process attempts to store data beyond the limit of fixed size buffer.
- Prevention by detecting and aborting
- Compile time defences – aim to harden programs to resist attacks in new programs
- Run-time defences - aim to detect and abort attacks in existing programs.

Week 7 Virtual Memory -

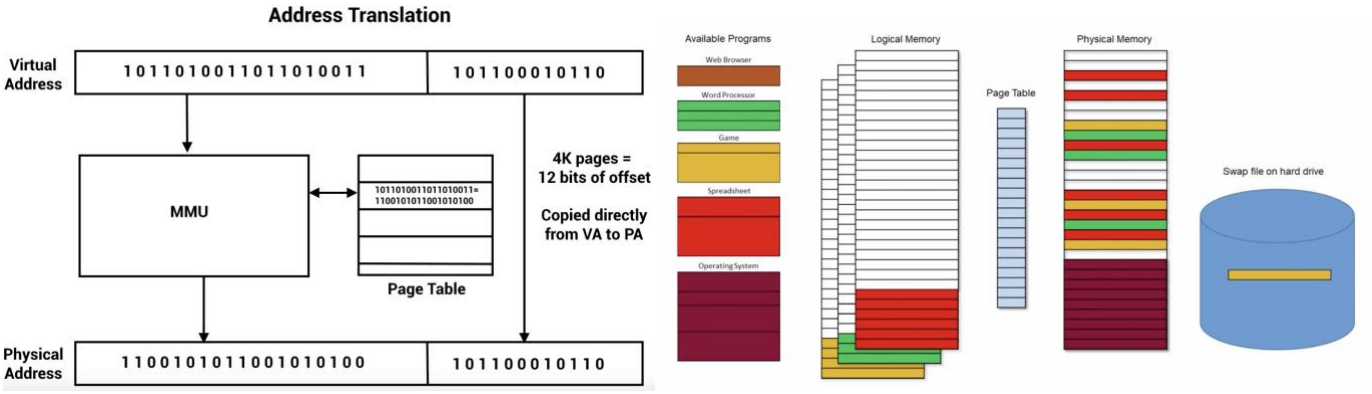
Segmented memory:

- Can't split processes memory, load it all or nothing.

- Segments are swapped between disc and main memory as needed; segments vary in size; the OS knows the start and size of each segment in physical memory.
- Each segment is atomic. Either whole segment is in RAM or none of the segment is in RAM.
- A segment in memory can only be replaced by a segment of the same size or smaller.
- Result in memory fragmentation – lots of small segments with gaps in between.
- Large segments may not be allowed into memory very often.
- Allows fast access

Paged memory:

- Memory is split up into small, equal sized sections called pages (or page frames)
- A single application may occupy multiple pages, which are not necessarily contiguous
- Each application program has its own view of the memory, known as logical memory
- A page table records where the different pages of a program are located in physical memory.
- Unused pages may be paged out to a swap file on disc to make room for others.



Virtual Memory

A storage allocation scheme in which secondary memory can be addressed as though it were part of main memory. The addresses a program may use to reference memory are distinguished from the address the memory system uses to identify automatically to the corresponding machine addresses.
The size of virtual storage is limited by the addressing scheme of the computer system and by the amount of secondary memory available and not by the actual number of main storage locations.

Virtual address

the address assigned to a location in Virtual Memory to allow that location to be accesses as though it were part of the main memory.

Virtual address space

virtual storage assigned to a process

Address space

range of memory addresses available to a process

Read address

address of a storage location in main memory.

Hardware and control structures

- All memory references are logical address that are dynamically translated into physical addresses at run time
- A process may be broken up into a number of pieces that don't need to be contiguously located in main memory during execution.

If two character presents, not necessary all of the pages or segments of a process in main memory during execution.

Execution of a process

- OS loads a few pieces of the program into main memory

- Resident set: portion of process in the main memory – currently resides in the main memory.
- An interrupt is generated when an address is needed that's not in the main memory.
- OS place the process in blocking state.
- Pieces of process containing the logical address is brought into main memory:
    - OS issues a disk I/O read request
    - Another process is dispatched to run while the disk I/O takes place
    - An interrupt is issued when disk I/O complete, causing the OS to take place the affected process in ready state.

Important conclusions:

- A process may be larger than all of main memory – OS automatically loads pieces of a process into main memory as required.
- More process may be maintained in main memory (partially loaded)

Locality

- Program and data references within a process tend to cluster
- Only a few pieces of a process will be needed over a short period of time
- Possible to make intelligent guesses about which piece will be needed in the future.

For virtual memory to be practical and effective: hardware must support paging and segmentation; OS must involve software for managing the movement of pages /segments between secondary memory and main memory.

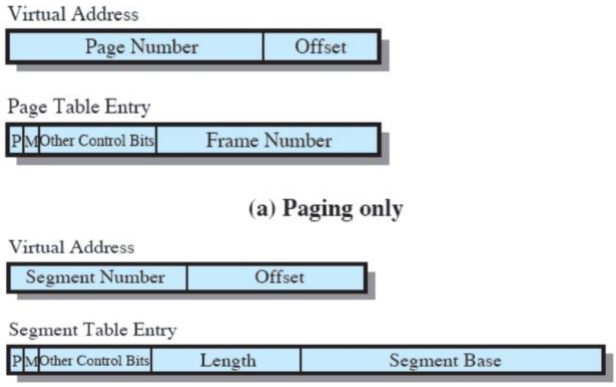
Simple Paging	Virtual Memory Paging	Simple Segmentation	Virtual Memory Segmentation
Main memory partitioned into small fixed-size chunks called frames		Main memory not partitioned	
Program broken into pages by the compiler or memory management system		Program segments specified by the programmer to the compiler (i.e., the decision is made by the programmer)	
Internal fragmentation within frames		No internal fragmentation	
No external fragmentation		External fragmentation	
Operating system must maintain a page table for each process showing which frame each page occupies		Operating system must maintain a segment table for each process showing the load address and length of each segment	
Operating system must maintain a free frame list		Operating system must maintain a list of free holes in main memory	
Processor uses page number, offset to calculate absolute address		Processor uses segment number, offset to calculate absolute address	
All the pages of a process must be in main memory for process to run, unless overlays are used	Not all pages of a process need be in main memory frames for the process to run. Pages may be read in as needed	All the segments of a process must be in main memory for process to run, unless overlays are used	Not all segments of a process need be in main memory for the process to run. Segments may be read in as needed
	Reading a page into main memory may require writing a page out to disk		Reading a segment into main memory may require writing one or more segments out to disk

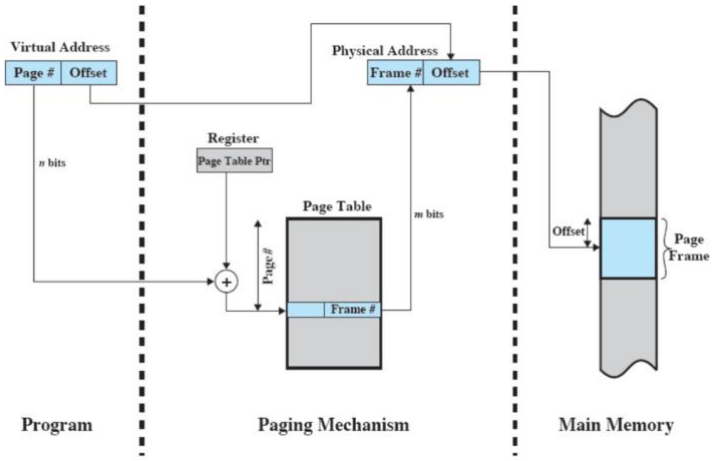
Paging

Each process has its own page table; each page table entry contains the frame number of the corresponding page in main memory.

Two extra bits are needed

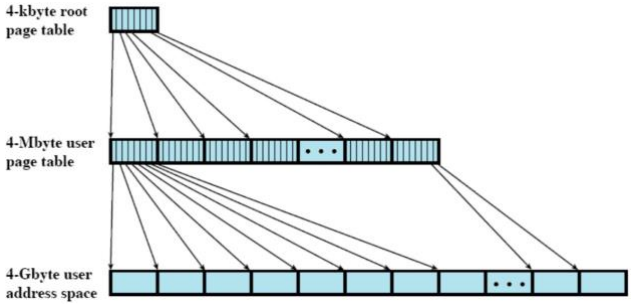
1. Whether it's in main memory or not
2. whether the contents of the page have been altered since it was last loaded.

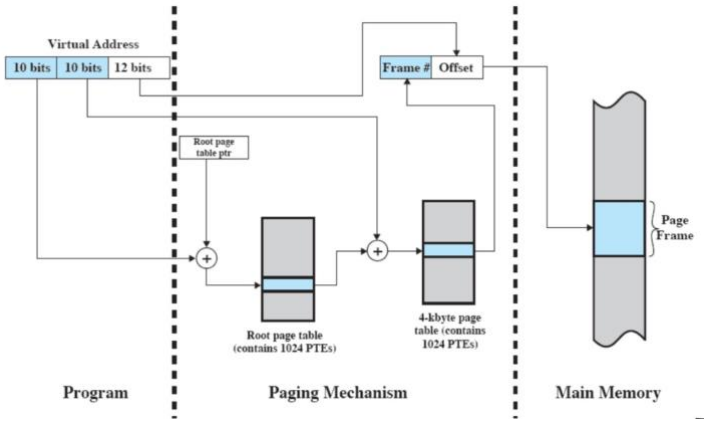




How to deal with really large page tables

We can page the page tables – to store page tables into virtual memory; when a process is running, part of its page table is in main memory.



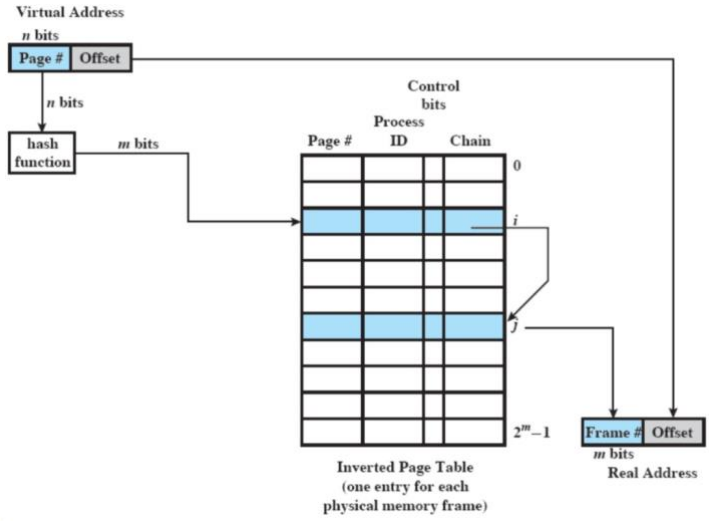


Or.. Inverted Page table

The page number portion of the virtual address is mapped into a hash value, and hash value points to inverted page table.

Fixed proportion of real memory is required for the table.

Indexes page table entries by frame number, not virtual page number.

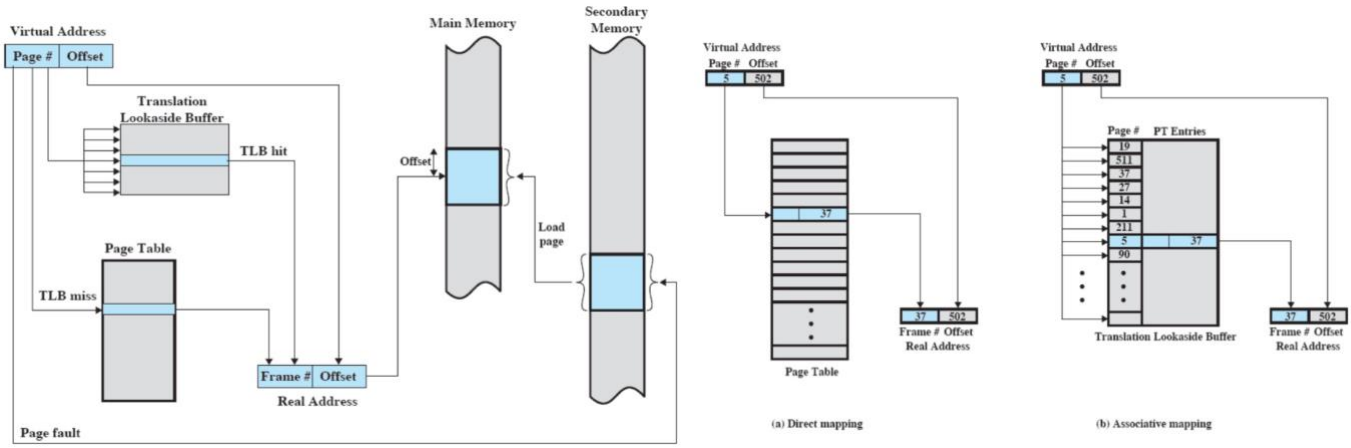


Page table entry includes:

1. Page number
2. Process ID
3. Control bits: includes flags, such as valid, referenced, etc.
4. Chain pointer: the index value of the next entry in the chain

Transition Lookaside Buffer

- Each virtual memory reference can cause two physical memory accesses – get and set table entry.
- Transition Lookaside Buffer is a high speed cache set up for page table entries – stores most recently used page table entries.
- Given a virtual address, if page table entry in TLB (hit), get frame number. Otherwise (miss), look up page table, check page in main memory or not, and TLB is updated to include the new page entry.



Associative look up: each TLB entry also includes the page number to directly index the frame number.

How to determine page size?

Smaller page size can get less amount of internal fragmentation.
But smaller page size leads to more pages required per process, hence larger page tables.
Larger page tables means **large portion of page tables in virtual memory. (page table could be paged again)**
For large programs in heavy multiprogrammed env, some portion of page table of active process must be in virtual memory.
most secondary memory devices **favour for larger page size** -> faster access to data.

Week 8 – virtual memory 2

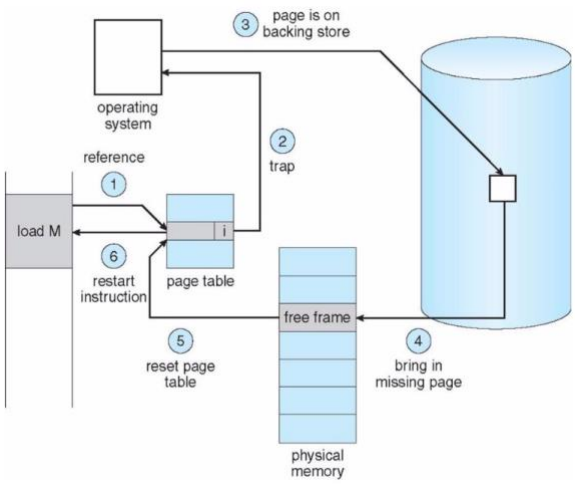
Why the page size is power of 2? Allows for successful location finding.
Because the virtual address is stored in binary numbers, power of two memory address can use the benefit of bit operations.

Valid-invalid bit – When the frame you're looking for is not in the main memory:

- With each page table entry a valid-invalid bit is associated. V = in memory. I = not in memory. Initially all the page table is I (because nothing has been visited yet)
- When doing address translation, if valid – invalid bit is I, means **page fault**

Page fault - What do we do if we send a instruction that caused a page fault?

1. **Find a free frame**
2. **Swap page into memory frame** via scheduled disk operation
3. **Reset tables to v**, show now it is in memory
4. **Restart the instruction** that caused the page fault.



Pre-paging vs. Demand Paging

Demand Paging (described above)

This method only brings pages into main memory when there is a reference made to location on the page. Therefore, we start with no pages in the memory, and there **will be many page faults when the process first started.**
A given instruction can access multiple pages and gets multiple page faults.

Prepaging

Pages other than one demanded are also brought in.
The method exploits the characteristic of most secondary memory devices. If pages of a process are stored contiguously in secondary memory, it is more efficient to bring in a number of pages at one time.
However, it is less effective if those extra pages are not referenced.

Replacement Policy

When all of the frames in main memory are **occupied** and it is necessary to bring in a new page, the replacement policy determines which page needs to be swapped out.

- Page removed should be the page least likely to be referenced in the near future. (following the principle of locality)
- Most policies predict future behaviour on the basis of past behaviour.

Basic Page replacement Procedure

1. use **modify bit** to reduce the number of times pages written back to disk.
2. Find location of the desired page on the disk
3. Find a free frame – if there is a free frame use it, otherwise use replacement algorithm. Write the victim frame back to disk if dirty.
4. Bring the new page into now free frame. Update page and frame tables.
5. Restarting the instruction

Frame Locking

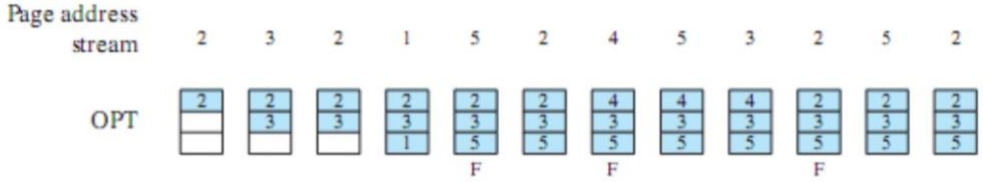
1. When a frame is locked, the page currently stored in that frame cannot be replaced.
2. Why locked? Kernel of the OS are locked. I/O bufferes and time-critical areas are locked. Locking is achieved by associating a lock bit with each frame.

Basic Replacement Algorithm

Optimal

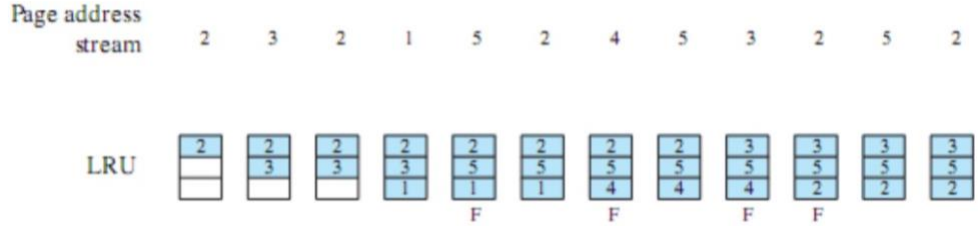
Optimal selects for replacement that page for which the time to the next reference is the longest.

- Impossible because we don't have information about future events.



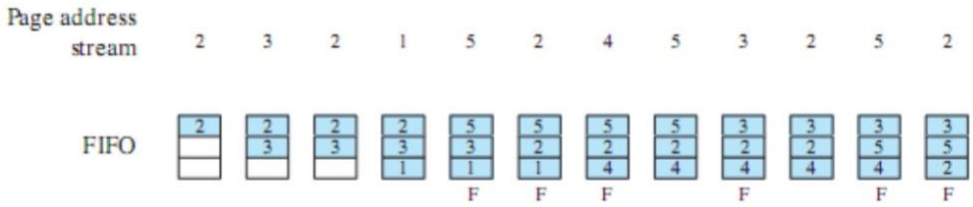
LRU (Least Recently Used)

Replaces the page that has not been referenced for the longest time. **By the principle of locality**, this should be the page least likely to be reference in near future.



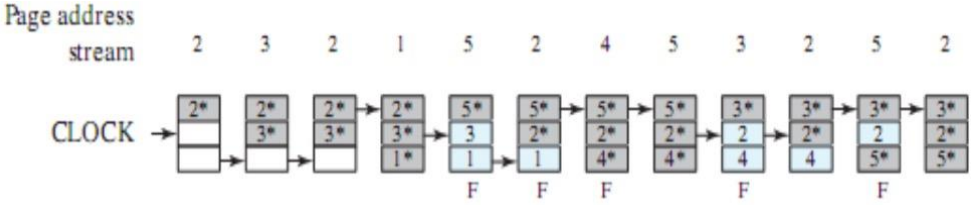
FIFO

Pages that has been in memory the longest is removed. (Not efficient because they're likely to be used again very soon)



Clock

Requires association of an additional bit with each frame.
When a page is first loaded/referenced, set bit to 1. If page fault happens, set all to 0.



Performance: OPT > LRU > CLOCK > FIFO.

Enhanced Clock Policy

Improve algorithm by using bit and modify bit. Take an ordered pair of **(use, modify)** to measure the desirability to replace.
(0,0) never used or modified – best to replace
(0,1) not used, but modified – must write out before replacement
(1,0) recently used but clean – probability will be used again soon
(1,1) recently used and modified – will be used again son and needs to write out before replacement.
The down side is it needs to search buffer several times.

Page buffering - simple FIFO policy.

Maintain two lists for replaced pages – Modified and Unmodified. Used as cache to the memory.

- Modified/Unmodified.
- Replaced pages will not be removed immediately from main memory but added to the tail of one of the lists.
- When a page fault occurs, the page brought into memory reads into unmodified free frame.
- When an unmodified page is to be replaced, it remains in the memory its page frame is added to tail of free page list.
- Significantly reduces I/O operation.

Resident Set Management

- Smaller amount of memory allocated to each process -> more process can fit in memory.
- Small number of pages -> increase page fault. Beyond a certain size, further allocations of pages will not affect the page fault rate.

Resident Set Management

Feature

	Local Replacement	Global replacement
Fixed Allocation	Number of frames allocated to a process is fixed Page to be replaced is chosen from among the frames allocated to that process	Not possible
Variable Allocation	Number of frames allocated to a process may be changed from time to time. Page to be replaced is chosen among frames allocated to that process	Page to be replaced is chosen from all available frames in main memory. Size of resident set varies. Free frame is added to resident set of process when a page fault occurs. If no free frame, replaces one from any process.

Pros Cons Requirements

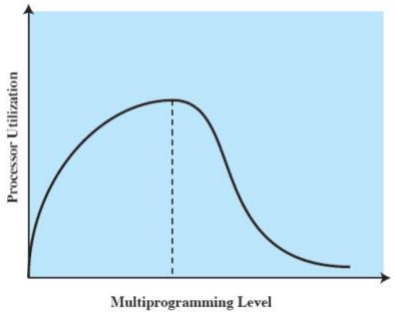
	Local Replacement	Global replacement
Fixed Allocation	Need to decide ahead of time the amount of allocation to give a process. If allocation too small, high page fault rate If allocation too large, too few programs in main memory.	Not possible
Variable Allocation	Dynamically re-evaluate allocation based on the assessment of the likely future demands of active process. Example: Variable Interval Sampled Working Set.	Easiest to implement. OS keeps list of free frames.

Cleaning Policy – replaced page back to secondary memory.

When should a replaced page write out to secondary memory?
Demand Cleaning: a page is written out only when it has been selected for replacement.
Pre-cleaning: pages are written out in batches.

Load control – Determines number of process will be resident in memory

- Determines the multiprogramming level
- Too few process – likely all processes will be blocked
- Too many process – thrashing. If too much multiprogramming, one/more of resident set process must be suspended. (swap out)



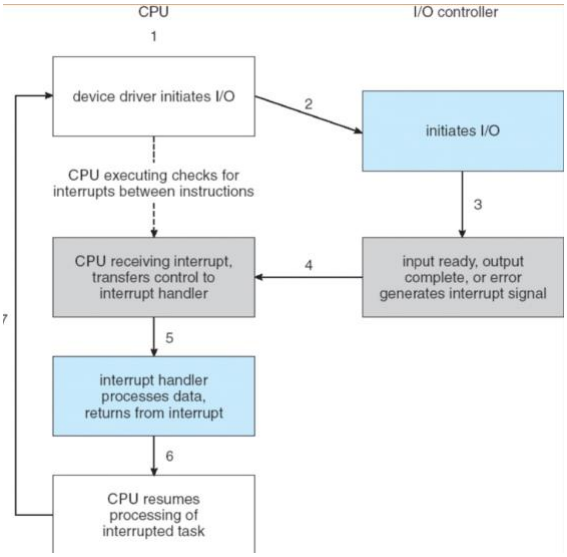
Linux Memory Management

- Process virtual memory – Multi level paging
  - **Page directory:** Process has a single page directory. Each entry points to one page of the **page middle directory.** Must be in main memory for active process.
  - **Page middle directory**: may span multiple pages. Each entry points to one page **in page table**
  - **Page table:** span multiple pages, each entry refers to one **virtual page of the process.**
  - Replacement uses clock algorithm. LFU (least frequently used) cache.
- Kernel memory allocation
  - **Dynamic allocation using a buddy system.**

Week 9 - I/O Management and Disk Scheduling

3 category of I/O devices: human readable, machine readable (like USB), Communication.
Three techniques to performing I/O

1. **Programmed I/O:** the programmer issues an I/O command on behalf of a process to an I/O module. That process then busy waits for the Operation to be completed before proceeding.
2. **Interrupt-driven I/O**: processor issues I/O command on behalf of a process.
   a. If **non-blocking**, processor continues to execute instructions from the process that issued the I/O command.
   b. If **blocking**, the next instruction the processor executes is from the OS, which puts the current process in blocked state and schedule another process.



3. **Direct Memory Access (DMA):** a DMA module controls the exchange of data between main memory and an I/O module. When data transfer completes, DMA module sends interrupt signal to processor.

Homework

Difference between page and page frame

- **Page**: in virtual storage, a fixed length block that has a virtual address and that is transferred as a unit between main / secondary memory.
- **Page frame:** a fixed size continuous block of main memory to hold a page.
- **Paging:** transfer of pages between main memory and secondary memory.

What is main memory and what is secondary memory?

Primary memory: where memory is stored temporarily, data is directly accessed by process, volatile because data cannot be retained when power failure, relatively faster than secondary memory, holds data currently being used by processing unit.

Secondary Memory: refer to external data storage where data is stored permanently. Data cannot be directly accessed. Non-volatile. Usually slower than main memory. Can hold substantial amount of data.

Homework 2

List 3 general category of information in a PCB. Give description for each category

Process Identification, Process state information, Process control information.

- 1 - Pid: unique identifier of this process.
- 2 - State information: describes its state in 5/7 state model, such as running, ready, suspended, etc.
- Memory block information: the MAB pointer
- Context information: used for recovery after interrupt and switching.
- Accountant information such as remaining CPU time
- Process priority.
- Data structuring – the next PCB it links to.
- The related I/O resources – resource ownership.
- **Process privileges** – the memory that may be accessed, types of instructions.
- **Interprocess communication.**

Describes the steps a kernel takes to contest switch between processes

1. Save the context of current process
2. Update current process PCB – to ready/blocked.
3. Move current process to appropriate queue
4. Select another process from the ready queue
5. Update new process PBC – state, context

6. Update memory requirement data structure
7. Restore context information in current PCB.

What is synchronous event and what is asynchronous event

A synchronous event occurs as the direct result of an instruction executed by the process.
E.g. If a process performs an illegal memory operation, the OS sends process a synchronous signal to abort.
An asyc event occurs due to an event unrelated to the current instruction, such signals must specify a process ID to indicate the signal recipient. Aysc signals are commonly used to notify I/O completion.

Briefly describe the 5-state processing model – each state

- Start: a job is admitted into the RR queue.
- Ready: when the job is loaded in the main memory and joined the round robin queue.
- Blocked: job is awaiting an event to be put back to Ready again
- Running: dispatched from ready queue and run by CPU
- Terminated/Exit: process finished its job or aborted and exits the executable processes main memory

What resources are used when a thread is crated? How does it differ from those used when a process is created?

- **Resources needed by process**:

- Allocating PCB
- Allocating memory space required by PCB, set memory map, list of open files, environment variables.
- Allocating and managing memory map is most costly step.

- **Resources needed by thread:**

- Only need to allocate a small data structure of a register set, stack, and priority.

A disadvantage of ULT is that when ULT executes a system, all threads within the process are blocked. Why?

- The ULT's structure is not visible to kernel – so kernel can only schedule them on Process basis!
- ULT shares the thread management data within user address space of a process. – so when one ULT is blocked that means all threads are set to this state.

Two programming examples of multithreading giving improved performance over single thread

- Two RPC call.
- Compute matrix that does not depends on each other.

Homework 3

What is busy waiting? What other kinds of waiting available in OS? Can busy waiting be avoided?

Busy waiting is when a program uses a while loop to repeating check some event occurred. This wastes CPU power because checking on a loop consumes CPU resource while the process doesn't do valuable things.
Other waiting is the "blocking" state of a process/thread. It can block on a condition and wait to be awakened in the future.
Busy waiting can be avoided by using monitor/condition variables.

Sleeping barber problem

The difference between Semaphore equals 0 or 1:
If the Semaphoe = 1, The first thread waiting for it will not be blocked, so wait gets executed before the signal.
If it is 0, signal goes first.

```
//one barber, one barber chair, number of chairs for waiting customers.

//if no customer, barber sits on his chair and sleeps

// if customer arrives, awakens barber. if there is available chair, sets on a chair. if all chairs are taken, leave.


semaphore Barber(0)

semaphore Customers(0)

semaphore AccessSeats(1) //mutex to protect seat number.
```

```
numberOfSeats = N


barber(){
    while(True){
        semWait(Customers) //if there is no customer, goes to sleep.
        semWait(AccessSeats) // there is availble customer. modify the number of seats.
        numberOfSeats+=1
        semSignal(AccessSeats) // unlock seats
        semSignal(barber) // tells a customer I'm available.
        service()
    }
}
customer(){
    semWait(AccessSeats); // checks seat number.
    if (numberOfSeats>0){
        numberOfSeats-=1
    }else{
        semSignal(AccessSeats) //unlock seat number protection
        exit()
    }
    semSignal(AccessSeats) //unlock seat number protection.
    semSignal(Customers) //Notify the arrival of a customer.
    semWait(Barber) //waits for barber to tell me I'm available
    //getting service
}
```

Show that a general counting semaphore can be implemented by binary semaphores.

```
binarySem mutex = 1
binarySem waiting = 0
semWaitc(int count){
    semWaitBinary(mutex)
    count--;
    if (count < 0){
        semSignalBinary(mutex); //place this process in s.queue
        semWaitBinary(waiting); //block this process
    }
    semSignalBinary(mutex);
}
semSignalC(int count){
    semWaitBinary(mutex)
    count++;
    if (count <= 0){
```

```
        semSignalBinary(waiting); //remove a process P from s.queue
    }
    semSignalBinary(mutex); //unloc
}
```

A file is to be shared among different processes, each of which has a unique number. File can be accessed simultaneously by several processes <= that number.

```
monitor file_access{
    access_limit = N
    current_access = 0
    Condition C
    void access_file(int my_num){ //once allowing multiple number of threads to access
        while(my_num+current_accese>=N){
            cwait(C)
        }
        current_access+=my_num
    }
    void finish_acess(int my_num){//once allowing multiple number of threads to access
        current_access-=my_num
        csignal(C)
    }
}
```

Homework 4

What is the difference between deadlock prevention and avoidance?

Deadlock prevention: (before finish writing the code) , adopt policy that eliminate one/more condition for deadlock
Deadlock avoidance: (after finish writing the code), add mechanism to make appropriate dynamic choices based on the current state of resource allocation.

Three processes share four resource units that can be reserved and released only one at a time. Each process needs a maximum of two units. Show deadlock cannot occur.

Deadlock happens when all four units are occupied while none of the process satisfies the condition to move on and release the resource.
When all 4 resources are occupied, at least one process gets 2 resources hence able to release the resource.

Comment on the following solution to the dining philosophers problem. A hungry philosopher first picks up his left fork; if his right fork is also available, he picks up the right fork and starts eating; otherwise he puts down his left fork again and repeats the cycle.

This is a good deadlock prevention strategy because it breaks the hold-and-wait condition. It is guaranteed the deadlock will not occur.
However, by keep trying to acquire the resource, computing resource is wasted. This solution leads to possible live lock because when all philosophers try to picking up at the same time, they will repeating try pickup and put down.

Homework 5

Consider the following set of processes, with the length of the CPU-burst time given in seconds.

Process	Arrival Time	Burst Time (cpu time)
P1	0	10
P2	2	1
P3	4	2
P4	6	1
P5	8	5

FCFS:

P1 start time = 0, finish time = 10, turnaround = 10, waiting = 0
P2 start time = 10, finish time = 11, turnaround = 9, waiting = 8
P3 start time = 11, finish time = 13, turnaround = 9, waiting = 7
P4 start time = 13, finish time = 14, turnaround = 8, waiting = 7
P5 start time = 14, finish time = 19, turnaround = 11, waiting = 6

SJF (shortest job first)

P1 start time = 0, finish time = 10, turnaround = 10, waiting = 0
P2 start time = 10, finish time = 11, turnaround = 9, waiting = 8
P4 start time = 11, finish time = 12, turnaround = 6, waiting = 5
P3 start time = 12, finish time = 14, turnaround = 10, waiting = 8
P5 start time = 14, finish time = 19, turnaround = 11, waiting = 6

(因为比较无聊 还没写)

Homework 6

In a fixed partitioning contiguous allocation scheme, what are the advantages of using unequal-size partitions?

Smaller pages can choose to fit in a smaller block, instead of fitting in a larger free block and causing more severe internal fragmentation.

Explain the difference between:

- Best fit policy: the memory manager chooses the block with most similar (but greater to) partition size to fit the current size requirement.
- Worst fit policy:
- First fit policy: the memory manager chooses the next block with greater or equal partition size.

Difference between internal and external fragmentation?

Internal fragmentation happens in fixed partitioning, where the process fitting in the memory block does not use all of the partition's memory.
External fragmentation happens in dynamic partitioning, where the memory space becomes fragmented because of swap in/out, leaving many small holes between partitions that are usually smaller than a typical memory requirement.

What is the distinction among logical, relative and physical address?

Logical(virtual) address is a reference to a memory location independent of the current assignment of data to a memory. With the help of logical – physical translation system, the logical address appears to be continuous for each individual process.
Relative address counts the offset – the address expressed as a location relative to some known point, usually the beginning of a program
Physical address is the actual location on the real main memory.

Why the page size set to be power of 2?

In a paging system, address translation requires that the virtual address be divided by the page size in order to determine the page number and the offset. If the address is already represented in binary, then the divide operations can be accomplished very quickly.

Homework 7

Suppose the virtual space accessed by memory is 6GB, the page size is 8KB, and each page table entry is 6 bytes. Computer the number of virtual pages that is implied. Also, compute the space required for the whole page table.

6GB = 6*2^30 bytes
8KB = 8*2^10 bytes -> page table size = 786432 entries.
Page table entry size = 6 bytes -> 4718592 bytes

What is the relationship between FIFO and clock page replacement algorithms?

- Clock page replacement algorithm is an improved version of FIFO that considers the used frequency of each pages/segments, performs better than FIFO.
- The simplest version of clock algorithm is like this:
  - when each page gets into main memory, mark the page entry with one bit.
  - When a page fault happens, all current one bit becomes 0.
  - Future page fault tries to select those with a bit marked as 0.
  - An improved clock version picks the victim based on both accessed/used bit and modified bit.

Why it is not possible to combine global replacement policy with fixed allocation policy?

- **Global replacement policy** means considering all unlocked frames to be a victim frame.
- **Fixed allocation policy** means allocating the same number of frames/pages in a process's lifetime.
- Fixed allocation policy requires when one page is swapped in, there must be other pages from this process to swap out – which is a local replacement policy, not global.

A computer provides VM space of 2^32 bytes, with **page size= 4096 = 2^12 bytes**.
The computer has 2^18 bytes of physical memory.
A user process generates virtual address 0x11123456. Explain how the system establishes the corresponding physical location.
'0001 0001 0001 0010 0011 **0100 0101 0110**'
The last 12 digits is in the displacement of the page table.
The other parts is the displacement of the entry in the page table.

Suppose a page reference stream: 2 3 4 3 2 4 3 2 4 **5** 6 7 5 6 7 **4** 5 6 7 2 1

FIFO

2	2	2	2	2	2	2	2	2	5	5	5	5	5	5	4	4	4	7	7	7
	3	3	3	3	3	3	3	3	3	6	6	6	6	6	6	5	5	5	2	2
		4	4	4	4	4	4	4	4	4	7	7	7	7	7	7	6	6	6	1
F	F	F							F	F	F				F	F	F	F	F	F

LRU

2	2	2	2	2	2	2	2	2	2	6	6	6	6	6	6	5	5	5	2	2
	3	3	3	3	3	3	3	3	5	5	5	5	5	5	4	4	4	7	7	7
		4	4	4	4	4	4	4	4	4	7	7	7	7	7	7	6	6	6	1
F	F	F							F	F	F				F	F	F	F	F	F

Review questions – chap 3,4,5,7,8

Chapter 5 – thread 伪代码题
Chapter 7, 8 – memory
Chapter 12 – memory manegment
每个 chapter 挑 5 题来做一下

Week 2
Self-revision questions

- Q1: Define process and explain the relationship between process and process control blocks
- Q2: Explain the concept of a process state and the state transitions

- Q3: List and describe the purpose of the data structures and data structure elements used by an OS to manage processes
- Q4: Understand the issues involved in the execution of OS code
- Q5: Assess the key security issues relating to OS
- Q6: Describe the process management scheme for UNIX SVR4