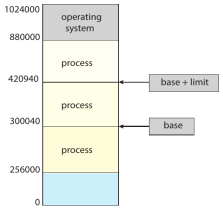
CS 415 Note Sheet

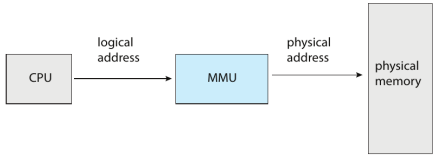
Main Memory

Definitions

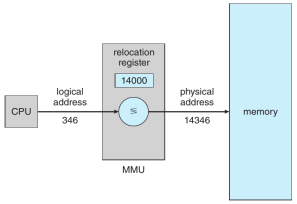
base register-holds the smallest legal memory address
limit register-specifies the range of memory space (total memory space = base + limit)
logical address-an address generated by the CPU
logical address space-set of all logical addresses generated
physical address-physical location on the memory unit, logical address becomes mapped to this
physical address space-physical address space is the set of all physical



addresses used
virtual address-address after binding stage (logical mapped to physical
memory management unit-maps virtual to physical addresses



relocation register-value is added to every addresss generated by program at the time that the address is sent to memory (ie if base is 14000, then accessing address 0 is dynamically relocated to location 14000, or accessing 346 gets moved to 14346)



dynamic loading-routines are not loaded into memory until they are called, useful in cases where routines are infrequently called (error checking)

Physical vs Logical memory

Logical Address is generated by CPU while a program is running. The logical address is virtual address as it does not exist physically, therefore, it is also known as Virtual Address. This address is used as a reference to access the physical memory location by CPU. The term Logical Address Space is used for the set of all logical addresses generated by a program's perspective. The hardware device called Memory-Management Unit is used for mapping logical address to its corresponding physical address.
Physical Address identifies a physical location of required data in a memory. The user never directly deals with the physical address but can access by its corresponding logical address. The user program generates the logical address and thinks that the program is running in this logical address but the program needs physical memory for its execution, therefore, the logical address must be mapped to the physical address by MMU before they are used. The term Physical Address Space is used for all physical addresses corresponding to the logical addresses in a Logical address space.

Differences Between Logical and Physical Address in Operating System

1. The basic difference between Logical and physical address is that Logical address is generated by CPU in perspective of a program whereas the physical address is a location that exists in the memory unit.
2. Logical Address Space is the set of all logical addresses generated by CPU for a program whereas the set of all physical address mapped to corresponding logical addresses is called Physical Address Space.
3. The logical address does not exist physically in the memory whereas physical address is a location in the memory that can be accessed physically.
4. Identical logical addresses are generated by Compile-time and Load time address binding methods whereas they differs from each other in run-time address binding method. Please refer this for details.
5. The logical address is generated by the CPU while the program is running whereas the physical address is computed by the Memory Management Unit (MMU).

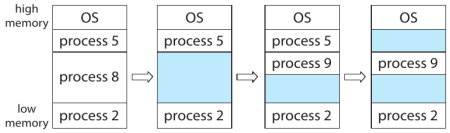
Swapping

Physical memory space is finite, and physical memory is allocated to meet the needs of a process. What happens if total physical memory space requested by processes exceeds physical memory?
A process can be swapped out of memory temporarily, moved to a disk large enough to hold copies of all memory images for all users. Later brought back for execution.
Swapping is used along with process scheduling.
Roll out, roll in is a swapping variant used with priority scheduling. Lower priority processes get swapped out so higher ones can execute.
Transfer time for swapping is directly proportional to the amount of memory swapped.
System maintains a ready queue of processes that have memory images on disk.

Allocation

Contiguous versus non-contiguous

Memory is divided into two paritions, OS and user. OS memory can be either the high chunk or the low chunk (Linux and Windows go high).
contiguous memory allocation-each process has a chunk of memory that is contiguous to the section containing the next process
variable-parition-allocating memory by assigning processes to variably sized partitions in memory, where each parition may contain a single process



When a process enters the system, OS checks to see if it has a memory block large enough for process. If it does not, it can either:

- Send and error and reject the process
- Place process into a queue to wait for a proper sized chunk

As memory is released, OS checks if it can allocate that memory to a waiting process. If a hole is too large, it gets split into two parts (holes are merged if adjacent). As process enter and leave memory, this can result in non-contiguous holes of various sizes in memory.
dynamic storage allocation problem-how to satisfy a request of size n from a list of free holes, many solutions exist:

- **first-fit**-iterate through holes and allocate first hole >= size n
- **best-fit**-iterate and allocate first hole = size n
- **worst-fit**-iterate and allocate largest hole of at least size n

First-fit and **best-fit** are **better** than worst fit in terms of speed and storage use

Algorithms

first-fit-Even with optization, given N blocks, another 0.5 N blocks will lost to fragmentation(third of memory becomes unusable **50-percent rule**)

Fragmentation

External

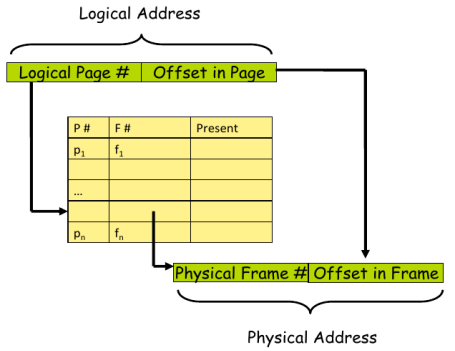
Both **first-fit** and **best-fit** suffer from **external fragmentation**.
external fragmentation-occurs when there is enough free memory space to satisfy a request, but available memory is not contiguous, it is broken into smaller pieces
compaction-shuffling memory contents so all the free blocks of memory are placed as a large block. This is not possible if relocation is static and done at assembly or load time. Compaction can only be done if relocation is dynamic and done at execution time.
Relocating dynamically only requires moving the program data and then adjusting its base register to reflect the new base address.
When compaction is possible, cost must be determined. A simple algorithm will move all processes to one end in memory, and all holes to the other, forming one large hole of memory. This can be expensive to run.

Internal

internal fragmentation-when breaking memory into fixed sized blocks, you allocate a block to a process that is bigger than its memory demand.
Difference in memory is **internal** to partition, but not being used.

Paging

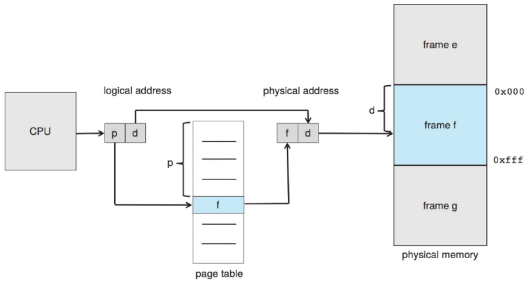
paging-memory management scheme that permits a processs's physical address space to be non-contiguous
Paging avoids external fragmentation and need for compaction, the two main problems with contiguous memory allocation. Paging works via cooperation between the OS and hardware.
frame-fixed size block of physical memory
page-fixed size block of logical memory
Every address generated by the CPU is broken into a **page number** and a **page offset**



page table-contains the base address of each frame in physical memory, and the offset is the location in each frame being referenced

Virtual-physical translation

- Programs are provided with a logical address space
- Role of the OS is to fetch data from either physical memory or disk (done by paging)
- Divide the logical addrress space into units called (logical) pages, each of which is of fixed size (usually 4k or 8k)
- Divide physical address space into (physical) frames



1. Extract page number p and use as index into the page table
2. Extract the corresponding frame number f from the page table
3. Replace the page number p in the logical address with the frame number f

The offset d does not change or become replaced, the frame number f and the offset now comprise the physical address.
Page size is defined by hardware (usually a power of 2 between 512 bytes and 16 Mbytes). Using a power of 2 allows easy translating from logical address to page number and page offset easy.
If logical address space is 2^m and page size is 2^n bytes, then the high order $m - n$ bits of a logical address designate the page number, and the n lower order bits of a logical address designate the page offset. So the logical address is as follows:

Page Map Example

- Given logical address $n = 2$ and $m = 4$.
- Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages).
- Logical address is page 0, offset 0
- Page 0 is in frame 5
- Thus logical address 0 maps to physical address 20 ($5x4 + 0$)
- Thus logical address 3 maps to physical address 23 ($5x4 + 3$)
- Logical address 4 is page 1, offset 0 and page 1 is mapped to fram 6
- Thus logical address 3 maps to physical address 24 ($6x4 + 0$)

Paging and Fragmentation

Paging has no external fragmentation, but can have internal (allocating an amount of pages equal to more memory than a program needs).

Internal Fragmentation Calculation

- Page size = 2048 bytes
- Process size = 72766 bytes
- 35 pages + 1086 bytes
- Internal Frag of 2048 - 1086 = 962 bytes
- Worst case frag = 1 frame - 1 byte
- Average case frag = 1/2 frame size

Are small frames desirable? Not really, becuase smaller frames need more page tables, and those take memory to track. Page sizes are growing over time.

frame table-system wide table detailing frames allocated to physical memory, and if it is free

Implementation of Page Table

Page table is kept in main memory

Translation Look Aside Buffer

Virutal Memory

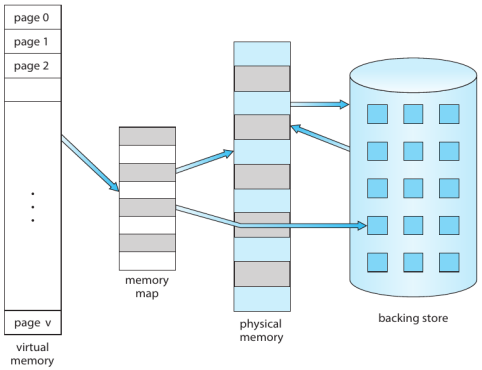
What is virtual memory

virtual memory-is the seperation of logical memory from physical memory. Virtual memory takes **program addresses** and **maps** them to **physical addresses**

Benefits

- Programs with memory needs larger than physical memory
- Allows sharing of address space by several processes
- More efficient process creation
- Less I/O needed to swap/load processses
- Libraries can be shared by processes through shared memory pages
- Processes can share memory regions
- Page can be shared during process creation with the fork() system call

virutal address space-refers to the logical/virtual view of how a process is stored in memory



demand paging-loading pages only when demanded by program execution

- Less I/O needed
- Less memory
- Faster reponse
- More users

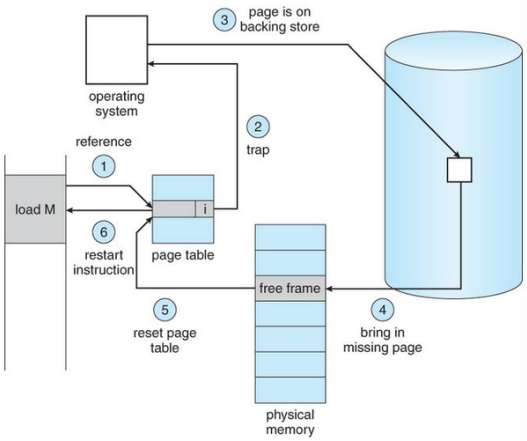
effective access time-allowing for 0 <= p <= 1 to be the probability of a page fault ($p = 0$ never faults, $p = 1$ always faults), and memory access time ma to be 10 nanoseconds;

- EAT = $(1 - p) * ma + p$*page fault time

Page fault handling

page fault-when a process tries to access a page that was not brought into memory

1. Service page fault interrupt
2. Read in the page
3. Restart the process



pure demand paging-processes are started with no pages in memory, only bringing them into memory after every page fault

free-frame list-pool of free frames for filling page fault request

zero fill on demand-frames are zeroed out before allocation, erasing contents (security issue)

1. Trap to OS
2. Save registers and process state
3. Determine if interrupt was a page fault
4. Check if page reference was legal, and determine location of page in the secondary storage
5. Issue a read from the storage to a free frame
    - Wait in queue for request to be serviced
    - Wait for the device seek/latency time
    - Begin transfer of page to free frame
6. While waiting, allocate CPU core to another process
7. Receive an interrupt from the storage I/O subsystem (I/O completed)
8. Save the registers and process state for process if step 6
9. Correct the page table and other tables to show that the desired page is now in memory
10. Wait for the CPU to be allocated to this process
11. Restore the register, process state, and new page table, then resume interrupted instruction

Performance estimations

Effective access time is directly proportional to the page fault rate

Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page fault service time = 8 ms
- EAT = $(1 - p) * 200 + p * 8000000$
  $= 200 + px7999800$
- If one access out of every 1000 causes a page fault, then EAT = $8.2us$
- Slowdown factor of 40
- For performance degredation below < 10 percent
  $220 > 200 + 7999800 * p$
  $20 > 7999800 * p$ $p < .0000025$
  < one page fault for every 400000 memory accesses

Demand Paging Opimizations

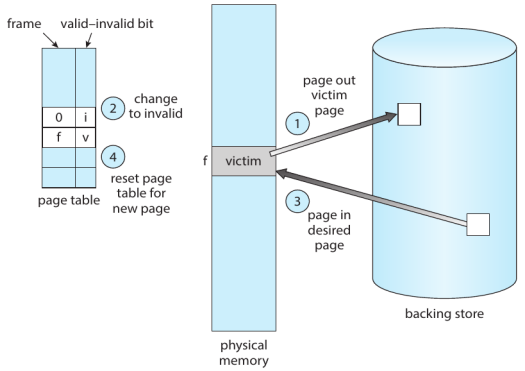
- Swap space I/O faster than file system I/O even on same device
  Swap sapce is allocated in larger chunks, less management needed
- Copy entire process image to swap space at process load time
  Then page in and out of swap space
- Demand page in from program binary on disk, but discard rather than paging out when freeing frame

Memory Initialization

Page Replacement

- Prevent over allocation of memory by modifying page fault routine to include page replacement
- Use **modify (dirty)** bit to reduce overhead of page transfer - only modified pages are written to disk
- Page replacement compeletes seperation between logical and physical memory - large virtual memory can be provided on a smaller physical memory

Basic Page Replacement



1. Find location of desired page on secondary storage
2. Find a free frame
    - If there is a free frame, use it
    - If there is no free frame, use page-replacement algorithm to select a **victim frame**
    - Write the victim frame to secondary storage (if needed); change the page and frame tables accordingly
3. Read the desired page into the newly freed frame; change the page and frame tables
4. Continue the process from where the page fault occured

If no frames are free, **two** page transfers (one for page out and one for page in) are required - increasing **EAT**.

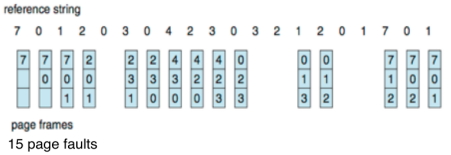
Overhead can be reduced by using a modify bit (or dirty bit). Each frame/page gets a bit asscociated with it in the hardware.

Page Replacement Algorithms

- Frame-allocation algorithm determines
  How many frames given to each process Which frames to replace
- Page-Replacement Algorithm
  Want lowest page-fault rate on both first access and re-access
- Evaluate algorithm by running on string of memory references, and computing number of page faults
  String is page numbers, not full addresses
  Repeated access to same page does not cause a page fault
  Results depend on number of frames available
- In all these examples, reference string is
  7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

FIFO

- 3 Frames (3 pages in memory at once per process)



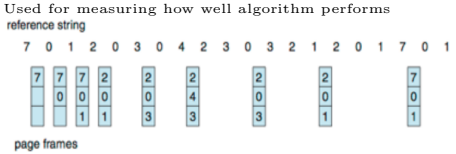
15 page faults

- Can vary by reference string: consider **1,2,3,4,1,2,5,1,2,3,4,5**
  Adding more frames can cause more page faults! (Belady's anomaly)
- How to track pages? Use a FIFO queue

Optimal Algorithm

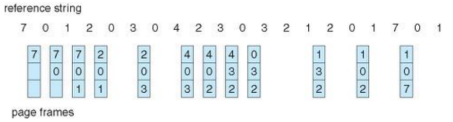
- Replace page that will not be used for longest period of time
  9 is optimal for example
- How do you know?

- Used for measuring how well algorithm performs



Least Recently Used

- Use past knowledge
- Replace page that has not been used in the most amount of time
- Associate time of last use with each page



- 12 faults - better than FIFO but worse than OPT
- Generally good, frequently used
- How to implement?
- Counter implementation
  Every page has a counter, every time a page is referenced through this entry, copy the clock into the counter
  When a page needs to be changed, check counters to find smallest value (search though table needed)
- Stack implementation
  Keep stack of pages in double link form
  Page referenced, move to top (6 pointers must be changed)
  Each update more expensive
  No search to replace
- LRU and OPT are cases of stack algorithms that don't have **Belady's Anomaly**
- LRU need special hardware and still slow
- Second chance algorithm
  Generally FIFO, hardware provided reference bit
  Clock replacement
  If page to be replaced has **ref bit = 0** then replace
  If **ref bit = 1** then set bit 0, leave page in memory

Enhanced Second Chance

- Improve by using reference bit and modify bit in concert
- Take ordered pair

  1. (0,0) not recently used not modified - best page to replace
  2. (0,1) not recently used but modified - not as good, must write out before replacement
  3. (1,0) recently used but clean, probably will be used again soon
  4. (1,1) recently used and modified - probably will be used again soon and need to write out before replacement

- When page replacement called for, use the clock scheme but use the four classes replace the page in lowest non-empty class
  Might need to search circular queue several times

Counting

Page Buffering

Belady's anomaly

Bélády's anomaly is the name given to the phenomenon where increasing the number of page frames results in an increase in the number of page faults for a given memory access pattern.
This phenomenon is commonly experienced in following page replacement algorithms:

1. First in first out (FIFO)
2. Second chance algorithm
3. Random page replacement algorithm

Reason of Belady's Anomaly – The other two commonly used page replacement algorithms are Optimal and LRU, but Belady's Anamoly can never occur in these algorithms for any reference string as they belong to a class of stack based page replacement algorithms.
A **stack based algorithm** is one for which it can be shown that the set of pages in memory for N frames is always a subset of the set of pages that would be in memory with N + 1 frames. For LRU replacement, the set of pages in memory would be the n most recently referenced pages. If the number of frames increases then these n pages will still be the most recently referenced and so, will still be in the memory. While in FIFO, if a page named b came into physical memory before a page – a then priority of replacement of b is greater than that of a, but this is not independent of the number of page frames and hence, FIFO does not follow a stack page replacement policy and therefore suffers Belady's Anomaly.

Number of frames = 3

Number of frames = 4

As the set of pages in memory with 4 frames is not a subset of memory with 3 frames, the property of stack algorithm fails in FIFO.

Why Stack based algorithms do not suffer Anomaly –
All the stack based algorithms never suffer Belady Anomaly because these type of algorithms assigns a priority to a page (for replacement) that is independent of the number of page frames. Examples of such policies are Optimal, LRU and LFU. Additionally these algorithms also have a good property for simulation, i.e. the miss (or hit) ratio can be computed for any number of page frames with a single pass through the reference string.

In LRU algorithm every time a page is referenced it is moved at the top of the stack, so, the top n pages of the stack are the n most recently used pages.Even if the number of frames are incremented to n+1, top of the stack will have n+1 most recently used pages.

Aspects of virtual memory

When to update: copy-on-write

- **copy-on-write**-allows both parent and child processes to intially share the same pages in memory
- COW allows more efficient process creation as only modified pages are copied
- In general, free pages are allocated from a pool of **zero-fill-on-demand** pages
- vfork() variation of fork() system call has parent suspend and child using COW address space of parent

Sharing: shared pages between processors

Use with I/O:memory mapped files

Thrashing

Working sets

File Systems

Definitions

File Systems Implementation

Definitions

Concepts

File system layers

Files, dirs, systems

Operations and usage

Remote file systems

Did not really cover.

Implementation

What is on disk? How is it formatted?

What is in memory? How is it represented?

Control blocks

System Usage

File allocation

Getting a file Caching

Free space management

Recovery

Did not really cover much.

I/O Systems

Definitions

Hardware

Devices

Controllers

Interfacing with Devices

Polling, interrupts

DMA

Types of I/O Operation

I/O system structure in OS

Processes and Threads

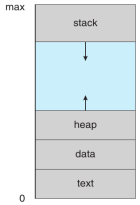


Figure 1: Process

What do the terms multiprogramming, multiprocessing, and multithreading mean? Is it possible to do multiprogramming if the computer system only has one processor?
multiprogramming-a computer running more than one processes at a time
multiprocessing-a computer using more than one CPU at a time
multithreading-having multiple threads of execution for one process
What is the purpose of system calls and how do they work? Give two examples of common system calls.
System calls allow the user to make a request for some type of service from the operating system. This allows for user programs to ask the OS to do some stuff on behalf of the program. It can also allow a process to send a request to the kernel. The kernel makes a range of services accessible to programs via the system call application programming interface (API). These services include, for example, creating a new process, performing I/O, and creating a pipe for inter process communication. A system call changes the processor state from user mode to kernel mode, so that the CPU can access protected kernel memory.
fork()-makes a new process (child process) that runs concurrently with the process making the **fork()** call. After the new child is created both processes execute the next instruction following the **fork()** system call. A child process uses the same pc, registers, and open files
wait()-Blocks the calling process until one of its child processes exits or recieves a signal. Child processes can terminate due to any of these: calls **exit()**, returns an int from main, or reiceives a signal whose default action is to terminate.
What is a process control block and what is its purpose?
The process control block contains pieces of information associated with a specific process.
Process State-state can be new, ready, running, waiting, halted, etc
Program Counter-counter includes the address of the next instruction to be executed
CPU registers-vary in number and type, such as accumulators, indexes, stack pointers general registers
Schedule info-process priority, pointers to schedule queue, and other parameters
memory-management information-information on base and limit registers, page tables, or the segment tables
accounting information-amount of CPU and real time used, time limits, account numbers, jobs, or process numbers
I/O status information-list of I/O devices allocated to the process, open files, etc
What is the process address space? What is in it and where are things located? Draw a picture to show this.
What is the difference between 1:1 M:1 threading? Why might we prefer a 1:1 threading model?
The one to one model maps each user thread to a kernel thread. Provides more concurrency than M:1, by allowing another thead to run when a thread makes a blocking system call. It also allows multiple threads to run in parallel on multiprocessors. The drawback to this model is that creating a user level thread requires creating a corresponding kernel thread, and a large number of kernel threads can burden the performace of a system. (Linux and Windows use 1:1)
Many to one maps many user level threads to one kernel level thread. Thread management is done by the thread library in the user space, so it is effiecient. However, the entire process whill block if a thread makes a blocking system call. Also because only one thread can access the kernel at a time, multiple theads are unabble to run in parallel on a multicore system.

Single thread vs multithreaded
Single threaded processes have their own code, data, files, registers, PC, and stack. The threads in a multithreaded process share code, data, and files, but have their own registers, stack, and PC.
What is a process? What is a thread? How do they differ? A process is a program in execution. A thread is a path of execution in a program.
1) Processes have their own memory space, threads have a shared memory space.
2) All the threads running within a process share the same address space, file descriptors, stack and other process related attributes.
3)It is termed as a 'lightweight process', since it is similar to a real process but executes within the context of a process and shares the same resources allotted to the process by the kernel.
OSC 4.4: What are two differences between user-level threads and kernel-level treads? Under what circumstances is one type better than the other?
User-level threads exist entirely at the user-level. That is, all of the common thread operations (e.g., creation, deletion, synchronization) are implemented in user code, typically a user library. The oper- ating system is not involved at all, and, in fact, is completely oblivious to the existence of user-level threads. The benefit of user-level threads is a closer coupling of thread interaction, as mainly evidenced by more rapid, finer-grained thread switching. Kernel-level threads, in contrast, are implemented in the OS kernel. System calls are necessary for thread creation, deletion, and so on. Once created, kernel-level threads have similar interaction capabili- ties as user-level threads. A significant difference exists, however, in how the threads execute. User-level threads share a single process. When the process blocks, none of the threads created as part of the process can execute. Threads supported by the kernel share the CPU. The OS has maintained enough state for each thread in a set of threads such that if one thread blocks, the other threads can continue to execute. Although the overhead of thread switching is greater with kernel-level threads, a greater degree of "real" concurrency is allowed.
User-level threads are often used to allow a program to perform logically concurrent parts of its com- putation without resorting to the creation of OS processes or kernel threads. When real concurrency is not required, user-level threads gives a low overhead way to switch between different threads of execution. It is also a way to achieve thread-based programming when the OS does not support threads, and user-level thread libraries can be ported to other architectures with only a rewrite of the thread switching code.
Kernel-level threads are "better" when one expects the individual threads to have OS involvement since the blocking of one thread (for example, due to a file I/O system call) does not automatically prevent the other threads from proceedings, as is the case with user-level threads. Even though the overhead of kernel- level thread switching is greater, the improved performance that comes from concurrent execution often more than makes up for it, especially when system call blocking is frequent. In reality, user-level threads must be "mapped" to kernel-level threads to actually run. In threading li- braries, such as Pthreads, there is this mapping taking place. It is possible in some systems to create more user-level threads than the number of kernel-level threads assigned to a process, thereby requiring the user- level thread scheduler to perform this mapping.
OSC 3.5: When a process creates a new process using the fork() operation, which of the following states is shared between the parent process and the child process: A. Stack, B. Heap, C. Shared memory segments. (Addition: Describe what problems might arise if the opposite were true.)
When a parent process uses fork() to create a child process, the child has its own memory space. In particular, the stack and heap part of that memory space are separate from parent (i.e., the addresses used to reference the stack and heap in the parent and child DO NOT go to the same physical location. However, if a shared memory segment is set up between the parent and the child, the memory segment is shared (i.e., the addresses used to reference the shared memory segment DO go to the same physical location). Instead, if it were the other way round and the stack and heap were shared by each child and parent process, they could theoretically overwrite one-another's data in memory. This would likely lead to unexpected and incorrect behavior that could cause the processes to function incorrectly.
What happens during a context switch?
When an interrupt occurs, the system needs to save the context of the processes running on the CPU, so that context can be restored.
This includes the value of the CPU registers, process state, and memory management info.
Generally, perform a save state of CPU (in kernel or user) and then save restore to resume.
Switching the CPU between processes is called a context switch. Context switching has a lot of overhead, because the system does no work while switching.
What is a process control block and what is its purpose?
It serves as a repository for all data needed to start or restart a proecess.
A process control block represents processes by storing relevant information, such as:
1) Process state: State may be new, ready, running, waiting, halted, etc
2) Program counter: Indicates address of the next instruction.
3) CPU Registers
4) CPU scheduling info: Process priority, pointers to schedule queues, other parameters.
5) Memory management info: Values for base and limit registers, page/segment tables, memory system
6) Accounting info: CPU/Real time used, time limits, account numbers, process numbers
7) I/O status: I/O devices allocated to process, list of open files, etc

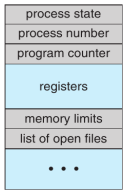


Figure 2: Philo

What do the fork(), exec(), and clone() system calls do?
fork(): Used to make new processes, that are the child of the caller. After fork call, BOTH processes will execute the next statement. Takes no arguments and returns a process ID.
exec(): Used to replace the process's memory space with a new program.
clone(): Makes threads.

Deadlocks

Concepts

What are the conditions for a deadlock to occur?
When a waiting thread never changes state because it is waiting for resources that are requested by other waiting threads.
In normal mode of operation, a thread may utilize a resource in only the following sequence:
1) Request: The thread requests the resource. If it cannot be granted immediately (such as if a mutex lock is currently held by another thread), then the requesting thread must wait until it can aquire the resource.
2) Use: The thread operates on the resource (such as a mutex lock, so the thread can access its critical section).
3) Release: The thread releases the resource.
Conditions for deadlock:
1) Mutual exclusion: At least one resource is hold in a non-sharable mode, only on thread at a time can uset the resource. If another thread requests that resource, the requesting thread must be delayed until the resource has been released.
2) Hold and wait: A thread must be holding at least one resource and waiting to aquire additional resources that are being held by other threads.
3) No preemption: Resources cannot be preempted, that is a resource can be released only by the thread holding it, after that thread has completed its task.
4) Circular wait: A set of waiting threads must exist such that

Synchronization

Concepts

race condition-a situation when several processses access and manipulate the same data concurrently
entry section-portion of code where a process requests entry into its critical section
exit section-portion of code where a process exits its critical section
What does mutual exclusion mean?
If a process is entering its critical section, then no other process can be executing their critical section.
What is a critical section? List the requirements for a solution to the critical section problem.
A critical section is a segment of code in which a process may be accessing/updating data that is shared by at least one other process. The critical section problem is to define a protocol that processes can use to sync activity and share data.
The requirements are:
1) Mutual exclusion: If a process is entering its critical section, then no other process can be executing their critical section.
2) Progress: If a process is executing in its critical section and some processes wish to enter their critical section, then only processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and selection cannot be postponed indefinitely.
3) Bounded Waiting: There exists a bound, or limit on the number of times that other processes can enter their critical sections after a process has made a request to enter its critical section and before that request is granted.
What is the difference between deadlock prevention and deadlock avoidance?
Deadlock prevention techniques do not allow (prevent) deadlocks from occurring by ensuring that at least oneof the necessary conditions cannot hold.
Deadlock avoidance techniques use information about the state of resource assignments and resource requeststo control resource allocation, thereby avoiding possible future deadlocks
preemptive kernel-allows a process to be preempted while it is running in kernel mode
nonpreemptive kernel-does not allow a process a process running in kernel mode to be preempted
What is the difference between busy waiting and blocking?
Busy waiting is a practice that allows a thread or process to use CPU time continuously while waiting for something. An I/O loop in which an I/O thread continuously reads status info while waiting for I/O to compile. Busy waiting wastes CPU cycles.
Blocking is where a process gives up the CPU and is wakened later when a condition is found to be true. Blocking does not use the CPU, where as busy waiting keeps it in a tight loop.
mutex lock-Mutual exclusion lock to protect critical sections and thus prevent race conditions. A process must aquire a lock before entering a critical section, and its releases the lock when when it exits its critical section.

```
while (true) {
acquire() lock
        critical section
release() lock
        remainder section
}

acquire() {
    while (!available)
        ; /* busy wait */
    available = false;
}

release() {
    available = true;
}
```

Semaphore-an integer variable, that apart from intialization, is accessed only through two standard atomic instructions: **wait()** and **signal()**

```
wait(S) {
    while (S <= 0)
        ; // busy wait
    S--;
}

signal(S){
   S++;
{
```

counting semaphore-can range over any unrestricted domain
binary semaphore-can range only from 0 to 1, behaves similarly mutex locks

Peterson's Solution

```
while (true) {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j)
        ;
        /* critical section */

    flag[i] = false;

    /*remainder section */
}
```

Mutual Exclusion-Note that each process enters its critical section only if either flag[j]==false or turn==i. Also note that if both processes can be executing their critical section at the same time, then flag[0] == flag[1] == true. This implies that both processes could not have successfully executed their while statements at the same time, since the value of turn can be either 0 or 1 but not both. Hence, one of the processes—say, Pj —must have successfully executed the while statement, whereas Pi had to execute at least one additional statement ("turn == j"). However, at that time, flag[j] == true and turn == j, and this condition will persist as long as Pj is in its critical section; as a result, mutual exclusion is preserved.
Progress and Bounded waiting-A process P can only be prevented from entering its critical section only if it is stuck in the while loop with condition flag[j] == true and turn == j; this loop is the only one possible. If Pj is not ready to enter its critical section, then flag[j] == false, and Pi can enter its critical section. If Pj has set flag[j] to true and is also executing in its while statement, then either turn == i or turn == j. If turn == i, then Pi will enter its critical section. If turn == j, then Pj will enter its critical section. However, once Pj exits its critical section, it will reset flag[j] to false, allowing Pi to enter its critical section. If Pj resets flag[j] to true, it must also set turn to i. Thus, since Pi does not change the value of the variable turn while executing the while statement, Pi will enter its critical section **(progress)** after at most one entry by Pj **(bounded waiting)**.

Mutex vs Semaphore

Consider the standard producer-consumer problem. Assume, we have a buffer of 4096 byte length. A producer thread collects the data and writes it to the buffer. A consumer thread processes the collected data from the buffer. Objective is, both the threads should not run at the same time.

Using Mutex

A mutex provides mutual exclusion, either producer or consumer can have the key (mutex) and proceed with their work. As long as the buffer is filled by producer, the consumer needs to wait, and vice versa.

At any point of time, only one thread can work with the entire buffer. The concept can be generalized using semaphore.

Using Semaphore

A semaphore is a generalized mutex. In lieu of single buffer, we can split the 4 KB buffer into four 1 KB buffers (identical resources). A semaphore can be associated with these four buffers. The consumer and producer can work on different buffers at the same time.

Misconception

There is an ambiguity between binary semaphore and mutex. We might have come across that a mutex is binary semaphore. But they are not! The purpose of mutex and semaphore are different. May be, due to similarity in their implementation a mutex would be referred as binary semaphore. Strictly speaking, a mutex is locking mechanism used to synchronize access to a resource. Only one task (can be a thread or process based on OS abstraction) can acquire the mutex. It means there is ownership associated with mutex, and only the owner can release the lock (mutex).

Semaphore is signaling mechanism ("I am done, you can carry on" kind of signal). For example, if you are listening songs (assume it as one task) on your mobile and at the same time your friend calls you, an interrupt is triggered upon which an interrupt service routine (ISR) signals the call processing task to wakeup.

General Questions

1. Can a thread acquire more than one lock (Mutex)?

Yes, it is possible that a thread is in need of more than one resource, hence the locks. If any lock is not available the thread will wait (block) on the lock.

2. Can a mutex be locked more than once?

A mutex is a lock. Only one state (locked/unlocked) is associated with it. However, a recursive mutex can be locked more than once (POSIX complaint systems), in which a count is associated with it, yet retains only one state (locked/unlocked). The programmer must unlock the mutex as many number times as it was locked.

3. What happens if a non-recursive mutex is locked more than once.

Deadlock. If a thread which had already locked a mutex, tries to lock the mutex again, it will enter into the waiting list of that mutex, which results in deadlock. It is because no other thread can unlock the mutex. An operating system implementer can exercise care in identifying the owner of mutex and return if it is already locked by same thread to prevent deadlocks.

4. Are binary semaphore and mutex same?

No. We suggest to treat them separately, as it is explained signalling vs locking mechanisms. But a binary semaphore may experience the same critical issues (e.g. priority inversion) associated with mutex. We will cover these in later article.

A programmer can prefer mutex rather than creating a semaphore with count 1.

5. What is a mutex and critical section?

Some operating systems use the same word critical section in the API. Usually a mutex is costly operation due to protection protocols associated with it. At last, the objective of mutex is atomic access. There are other ways to achieve atomic access like disabling interrupts which can be much faster but ruins responsiveness. The alternate API makes use of disabling interrupts.

6. What are events?

The semantics of mutex, semaphore, event, critical section, etc. . . are same. All are synchronization primitives. Based on their cost in using them they are different. We should consult the OS documentation for exact details.

7. Can we acquire mutex/semaphore in an Interrupt Service Routine?

An ISR will run asynchronously in the context of current running thread. It is not recommended to query (blocking call) the availability of synchronization primitives in an ISR. The ISR are meant be short, the call to mutex/semaphore may block the current running thread. However, an ISR can signal a semaphore or unlock a mutex.

8. What we mean by "thread blocking on mutex/semaphore" when they are not available?

Every synchronization primitive has a waiting list associated with it. When the resource is not available, the requesting thread will be moved from the running list of processor to the waiting list of the synchronization primitive. When the resource is available, the higher priority thread on the waiting list gets the resource (more precisely, it depends on the scheduling policies).

9. Is it necessary that a thread must block always when resource is not available?

Not necessary. If the design is sure 'what has to be done when resource is not available', the thread can take up that work (a different code branch). To support application requirements the OS provides non-blocking API.

For example POSIX pthread-mutex-trylock() API. When mutex is not available the function returns immediately whereas the API pthread-mutex-lock() blocks the thread till resource is available.

Dining Philosopher

The Dining Philosopher Problem – The Dining Philosopher Problem states that K philosophers seated around a circular table with one chopstick between each pair of philosophers. There is one chopstick between each philosopher. A philosopher may eat if he can pickup the two chopsticks adjacent to him. One chopstick may be picked up by any one of its adjacent followers but not both.

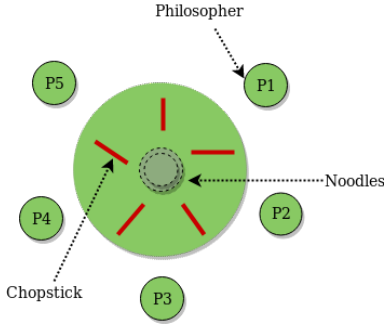


Figure 3: Philo

Each philosopher is represented by the following pseudocode:

```
process P[i]
 while true do
   {  THINK;
      PICKUP(CHOPSTICK[i], CHOPSTICK[i+1 mod 5]);
      EAT;
      PUTDOWN(CHOPSTICK[i], CHOPSTICK[i+1 mod 5])
   }
```

There are three states of philosopher : THINKING, HUNGRY and EATING. Here there are two semaphores : Mutex and a semaphore array for the philosophers. Mutex is used such that no two philosophers may access the pickup or putdown at the same time. The array is used to control the behavior of each philosopher. But, semaphores can result in deadlock due to programming errors.