Overview

System Programs: associated with the operating system but are not necessarily part of the kernel.
Middleware: software frameworks that provide additional services to application developers.

Interrupts

Signal to CPU there is a task requiring attention, usually I/O. Trap is s/w generated interrupt caused by error or user request.
Synchronous: user process waits for I/O to finish, must know time it takes. **Asynchronous:** request returns before I/O completion, interrupts when done.
Polling system: system calls all interrupt handlers to determine which made the interrupt. (slow) **vectored system:** handler is found via interrupt vector.
Interrupt is maskable if it can be ignored by CPU otherwise nonmaskable.

I/O Techniques

Programmed: Requested I/O action performed, module sets I/O status register. Processor periodically checks status until action completed.
Interrupt-driven: Process issues I/O command to module which interrupts processor when done. Processor then transfers data.
DMA: Data is transferred directly to memory, processor only needed for initialization. Bus access slowed for processor during the transfer.

Processes

Possible states of a process: new, running, waiting, ready, terminated.

Process Creation Code

```
int main()
{
    pid_t pid;
    /* fork a child process */
    pid = fork();
    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) { /* child process
        ↪ */
        execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
        /* parent will wait for the child
            ↪ to complete */
        wait(NULL);
        printf("Child Complete");
    }
    return 0;
}
```

`Fork()` returns 0 for the child process, and returns the `pid` of the child process for the parent. The `exit(1)` system call can be used to terminate a process with status 1. The parent can obtain the status of a terminated child with `pid = wait(&status)` where `status` is the integer status of the child process. A process that has terminated, but whose parent has not yet called `wait()`, is a **zombie** process. If the parent of a process terminated without invoking `wait()`, the process is an **orphan**.

Posix Shared Memory

```
/* create the shared memory object */
fd = shm_open(name,O_CREAT | O_RDWR,0666)
    ↪ ;
/* configure the size of the shared
    ↪ memory object */
ftruncate(fd, SIZE);
/* memory map the shared memory object */
ptr = (char *)
mmap(0, SIZE, PROT_READ | PROT_WRITE,
    ↪ MAP_SHARED, fd, 0);
/* write to the shared memory object */
sprintf(ptr,"%s",message_0);
ptr += strlen(message_0);
sprintf(ptr,"%s",message_1);
ptr += strlen(message_1);
/* read from the shared memory object */
printf("%s",(char *)ptr);
/* remove the shared memory object */
shm_unlink(name);
```

Pipes (message passing)

```
#define BUFFER_SIZE 25
#define READ_END 0
#define WRITE_END 1
int main(void)
{
    char write_msg[BUFFER_SIZE] = "
        ↪ Greetings";
    char read_msg[BUFFER_SIZE];
    int fd[2];
    /* create the pipe */
    if (pipe(fd) == -1) {
        fprintf(stderr,"Pipe failed");
        return 1;
    }
    /* fork a child process */
    pid = fork();
    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    if (pid > 0) { /* parent process */
        /* close unused end of pipe */
        close(fd[READ_END]);
        /* write to pipe */
        write(fd[WRITE_END], write msg,
            ↪ strlen(write msg)+1);
        /* close write end of pipe */
        close(fd[WRITE_END]);
    }
    else { /* child process */
        /* close unused end of pipe */
        close(fd[WRITE_END]);
        /* read from pipe */
        read(fd[READ_END], read msg,
            ↪ BUFFER SIZE);
        printf("read %s",read msg);
        /* close read end of pipe */
        close(fd[READ_END]);
    }
    return 0;
}
```

Threads

Amdahl's Law

$speedup \leq \frac{1}{S+(1-S)/N}$ where S is the serial portion of the application and N is the number of processing cores
Threads of a process share the code section, data section, and files of the process. The registers, stack, and program counter of the threads differ.
Data Parallelism: distribute subsets of the same data across multiple computing cores.
Task Parallelism: distribute tasks (threads) across multiple computing cores.
Five areas of challenge in multicore programming: identifying tasks, balancing the amount of work tasks do, data splitting, data dependency, and testing and debugging.
Synchronous Signals: Delivered to the same process that caused the signal.
Asynchronous Signals: Delivered to a process that did not cause the signal.
Thread Local Storage (TLS): Data local to a specific thread. Similar to the concept of static data and local variables.
Lightweight Processes (LWP): Intermediate data structure between user and kernel threads often used in systems implementing many-to-many or two-level thread models. Looks like a virtual processor to the user-thread library. The application can schedule a user thread to run on an LWP. Each LWP is attached to a kernel thread, and it is kernel threads that the operating system schedules to run on physical processors. If a kernel thread blocks, the LWP blocks and also blocks the user thread down the chain.
Benefits of Threads: Responsiveness, resource sharing (threads share data by default), economy (threads are take less time and memory to create threads then processes), scalability.

Pthread

```
pthread_t tid; /* thread identifier */
pthread_attr t attr; /* set of thread
    ↪ attributes */
/* set default attributes of thread */
pthread_attr_init(&attr);
/* create the thread */
pthread_create(&tid, &attr, runner, argv
    ↪ [1]);
/* cancel the thread */
pthread_cancel(tid);
/* check if there is a cancellation
    ↪ request */
pthread_testcancel();
/* wait for the thread to exit */
pthread_join(tid,NULL);
pthread_exit(0);
```

Synchronization

Critical section problem

A solution to the critical section problem must satisfy mutual exclusion, progress (processes are eventually allowed to access their critical sections after requesting it), and bounded waiting (there is a limit on the number of times other processes can access their critical sections after one process has requested access) **Peterson's Solution:** processes use a flag variable to indicate if a process is ready to enter its critical section and a shared turn variable to indicate which process' turn it is to enter their critical section.

Semaphores

`wait()` = `P()`, `signal()` = `V()`, `wait()` decrements the semaphore, `signal()` increments the semaphore.

Atomic Instructions

```
boolean test_and_set(boolean *target) {
    boolean rv = *target;
    *target = true;
    return rv;
}
do {
    while (test_and_set(&lock))
        ; /* do nothing */
        /* critical section */
    lock = false;
        /* remainder section */
} while (true);
int compare_and_swap(int *value, int
    ↪ expected, int new_value) {
    int temp = *value;
    if (*value == expected)
        *value = new_value;
    return temp;
}
while (true) {
    while (compare_and_swap(&lock, 0, 1)
        ↪ != 0)
        ; /* do nothing */
        /* critical section */
    lock = 0;
        /* remainder section */
}
```

Liveness: A set of properties that a system must satisfy to ensure that processes make progress during their execution life cycle.

Pthread Synchronization

Mutex Locks

```
pthread_mutex_t mutex;
/* create and initialize mutex lock */
pthread_mutex_init(&mutex,NULL);
/* acquire the mutex lock */
pthread_mutex_lock(&mutex);
/* critical section */
/* release the mutex lock */
pthread_mutex_unlock(&mutex);
```

Condition Variables

```
pthread_mutex_t mutex;
pthread_cond_t cond_var;
pthread_mutex_init(&mutex,NULL);
pthread_cond_init(&cond_var,NULL);
// thread waiting on condition variable
pthread_mutex_lock(&mutex);
while (a != b)
    pthread_cond_wait(&cond_var, &mutex);
pthread_mutex_unlock(&mutex);
// thread that modifies shared data,
    ↪ signalling waiting thread
pthread_mutex_lock(&mutex);
a = b;
pthread_cond_signal(&cond_var);
pthread_mutex_unlock(&mutex);
```

Deadlocks

Is Deadlock Possible

Necessary Conditions

Mutual Exclusion - A resource can only be held by one process at a time. **Hold and Wait** - A process can hold a number of resources at a time and at the same time, it can request for other resources that are being held by some other process. **No preemption** - A resource can't be preempted from the process by another process, forcefully. **Circular Wait** - There is a circular dependency of resources that processes are waiting for.

Reader-writer locks

If only reader-writer locks are used in a multithreaded application, then deadlock is still possible.
Locks cannot be shared if there is a writer(mutual exclusion).
A thread can hold one reader-writer lock while waiting to acquire another(hold and wait).
You cannot take a lock away, so no preemption is upheld.
A circular wait among all threads is possible.

Banker's Algorithm

Need = Max - Allocation.
Safe state = all threads can finish executing with the available resources.
If need < available for a particular resource for all (remaining) threads then not in safe state.
Otherwise execute the threads that can be executed until no threads remain.

Resource Allocation Graphs

A directed edge from thread T_i to resource type R_j is denoted by $T_i \rightarrow R_j$; and signifies that thread T_i has requested an instance of resource type R_j.
A directed edge from resource type R_j to thread T_i is denoted by $R_j \rightarrow T_i$ signifies that an instance of resource type R_j has been allocated to thread T_i.

Methods for Handling Deadlock

Deadlock Prevention: prevent circular wait from happening (only practical solution)
Deadlock Avoidance: don't grant resources if it leads the system to an unsafe state.
Deadlock Detection and Recovery: Either terminate or preempt resources.

Scheduling

Predicting next CPU burst Formula

$\tau_{n+1} = \alpha \cdot t_n + (1-\alpha) \cdot \tau_n$ where t_n is the value of the nth CPU burst, $0 \leq \alpha \leq 1$
τ_0 affects the starting value of the predictions, and α affects how much the last CPU burst vs the last predicted CPU burst is weighted.
If $\alpha = 0$, then $\tau_{n+1} = \tau_n$ and recent history has no effect on the future CPU burst. If $\alpha = 1$, then $\tau_{n+1} = t_n$ and only the most recent CPU bursts matter (history is assumed to be old and irrelevant). $\alpha = 1/2$ weights recent and past history evenly.

Scheduling Algorithms Formulas

Throughput: Number of processes completed per time unit **Turnaround time of a process** is difference between when the process finishes execution and its arrival time; turnaround time = process finish time - start time. **Waiting time of a process** is how long a process does not execute on the CPU from its arrival. The time a process spends waiting on the ready queue. **CPU utilization rate** is the time the CPU spends executing processes divided by the total time (time spent executing + time spent idle) **Response Time:** The difference between the time a process produces a response and the time of the submission of a request (possibly same as arrival time?)
SJF cannot be implemented at the level of CPU scheduling because you cannot know the length of the next CPU burst (you can't see the future).
For **round robin scheduling**, the time quantum should be large with respect to the context switch time so not too many context switches are done, but not too large so as to devolve to FCFS scheduling.

Chip Multithreading

When you have more than one hardware thread in a core. The threads look like two logical CPUs to the OS.
Coarse grained multithreading: a thread executes on a core until a long-latency event such as a memory stall occurs. Because of the delay caused by the long-latency event, the core must switch to another thread to begin execution. However, the cost of switching between threads is high, since the instruction pipeline must be flushed before the other thread can begin execution on the processor core.
Fine grained multithreading: threads switch at the boundary of an instruction cycle. Fine-grained systems include logic for thread switching, so the cost of switching between threads is small.

Load Balancing

Push migration: a specific task periodically checks the load on each processor and if it finds an an overloaded processor it evenly distributes the load by pushing/moving threads to idle or less-busy processors.
Pull migration: when an idle processor pulls a waiting task from a busy processor.

Processor Affinity

A process has an "affinity" for the processor it is currently running on because the process will usually have values it uses often in the cache of the current processor.
Soft Affinity: OS will attempt to keep a process on a single processor, but it is possible for a process to migrate between processors during load balancing. **Hard Affinity:** OS allows a process to specify a subset of processors on which it can run using sys calls.

Real time Scheduling

The scheduler for a real-time operating system must support a priority based algorithm with preemption
Event latency: amount of time that elapses from when an event occurs to when it is serviced.
Interrupt latency refers to the period of time from the arrival of an interrupt at the CPU to the start of the routine that services the interrupt.
Dispatch latency amount of time required for the scheduling dispatcher to stop one process and start another.
Little's Formula: $n = \lambda \cdot W$ where n is the average long-term queue length (excluding the process being serviced), λ is the average arrival rate for new processes in the queue, and W is average long-term queue length.

Gantt Chart

Process	Burst Time
P_1	24
P_2	3
P_3	3

Suppose that the processes arrive in the order: P_1, P_2, P_3 The Gantt Chart for the schedule is:

Memory Management

Dynamic loading: a routine is not loaded until it is called. All routines are kept on disk in a relocatable load format. The main program is loaded into memory and is executed. When a routine needs to call another routine, the relocatable linking loader is called to load the desired routine into memory if its not loaded and to update the program's address tables to reflect this change. Then control is passed to the newly loaded routine.
Dynamic Linked Libraries are similar.

Fragmentation

First fit strategy given N blocks will lose 0.5N blocks to fragmentation; 1/3 of memory will be unusable. Is also called the 50% rule.
Compaction is only possible if relocation is dynamic and done at runtime, not possible if relocation is static and done at assembly or load time.

Paging

If the size of the logical address space is 2m, and a page size is 2n bytes, then the high-order m-n bits of a logical address designate the page number, and the n low-order bits designate the page offset.
If process size is independent of page size, we expect internal fragmentation to average one-half page per process.
The **frame table** has one entry for each physical page frame, indicating whether the latter is free or allocated and, if it is allocated, to which page of which process (or processes). Used by the OS to keep track of physical memory.
Reentrant code is non-self-modifying code: it never changes during execution and can be shared among processes. Can be implemted as shared pages.

Number of Bits and/or Entries

Important: if any size is not in bytes convert to bytes first.
If the address takes x bits to store, page size is 2^y bytes, and number of frames or pages is 2^z, then $x = y + z$.
Conventional single level page table stores an entry for each virtual page or page number. **Inverted page table** stores an entry for each physical frame.

Internal Fragmentation Formula

Given page size = p, process size r, pages used = n, fragmentation = $n \cdot p - r$

Virtual Memory

Demand Paging

effective access time = $(1-p) \cdot ma + p \cdot$ page fault time where p is the probability of a page fault and ma is the memory access time.
A **valid/invalid bit** is used to tell a process if a page is legal and in memory or if its invalid (not in address space of the process) or legal but not in memory.
OS' maintain a **free frames list** which is a pool of free frames for satisfying requests for new pages being brought into memory.
Zero-fill-on-demand: the contents of frames are erased before being allocated.

File System

Volume: disk block that contains file system.
Info associated with an open file: file pointer, file-open count, location of the file, and access rights to the file.
Operations to be performed on a directory: search for a file, create a file, delete a file, list a directory, rename a file, traverse the file system.

Directory Structure

Single-level directory: Only one directory exists for all users and all files are kept in it.
Two-level directory: Each user has their own directory and a master file directory is indexed by username to point to the user's directory. Isolates users from each other. Is a tree of height 2.
Garbage collection involves traversing the entire file system, marking everything that can be accessed. Then, a second pass collects everything that is not marked onto a list of free space. Only used in general graph directory structures.
Logic for "answering maximum size of file" type questions: multiply the number of direct disk blocks by size of the disk blocks, for indirect disk blocks, divide the size of the disk blocks by the size of a pointer to a disk block, then raise this number by a power which is the same as the level of the indirect block (i.e. for single indirect raise by a power of 1, for a double indirect raise by a power of 2, and so on). Multiply this value by the size of a disk block and by the amount of that specific indirect disk block, i.e. if you are doing the calculation for 5 single indirect disk blocks, then multiply by 5 at the end. Add the multiplications for all the types of disk blocks together for the final answer.

Mass Storage Systems

SCAN Scheduling

The disk arm starts at one end of the disk and moves toward the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk. At the other end, the direction of head movement is reversed, and servicing continues

C-SCAN Scheduling

C-SCAN moves the head from one end of the disk to the other, servicing requests along the way. When the head reaches the other end, however, it immediately returns to the beginning of the disk without servicing any requests on the return trip. Is a variant of SCAN scheduling.

Write amplification: The creation of I/O requests not by applications but by the NVM device doing garbage collection and space management. Can greatly impact the write performance of the device.

latency & I/O time

Latency based on spindle speed: 1/(RPM/60) = 60/RPM

Average latency = 1/2 latency

Access Latency = average seek time + average latency

Average I/O time = access latency + (amount to transfer / transfer rate) + controller overhead

Conversions

1 s = 1000 ms, 1 ms = 1000 µs (microsecond), 1 µs = 1000 ns

1 kB = 2^{10} B = 1024 B,

1 MB = 2^{20} B = 1,048,576 B,

1 GB = 2^{30} B = 1,073,741,824 B,

1 TB = 2^{40} B = 1,099,511,627,776 B