Operating System

- Program that acts as an intermediary between a computer user and the computer hardware
- Must run in kernel mode

No OS	Minimal overhead, not portable, inefficient use of computer
Batch	Execute job sequentially, inefficient use of CPU
Time-sharing	Users use terminals to schedule jobs
Personal	Dedicated to user

Benefits:

1. Abstraction: same API to access all types of same category of hardware
2. Resource allocator: manages all resources like CPU, memory, I/O and arbitrate potentially conflicting requests for efficient and fair resource use
3. Control program: controls execution of programs, preventing errors and improper use

Kernel

- Deals with hardware issues, provides system call interface, and special code for interrupt handlers and device drivers

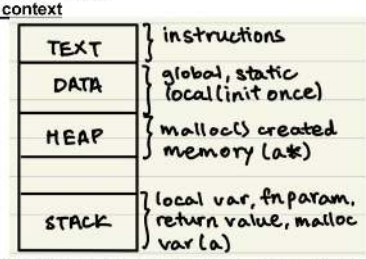
Type	Description	Pros	Cons
Monolithic	Single program; does everything for OS to handle user <> hardware	Well understood and good performance	Highly coupled components, complex structure
Microkernel	Only implements essential functionality like IPC; higher-level services run outside of kernel and use IPC	Robust and extendible; better isolation and protection	Lower performance

Virtual machines

- Software emulation of hardware, virtualizing the underlying hardware
- Managed by hypervisor

Hypervisor (Virtual Machine Monitor): creates and manages virtual machines
Type 1: bare metal (i.e. hypervisor is lightweight OS to create virtual machines)
Type 2: run on host OS

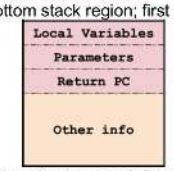
Memory context



Stack: information for function invocation in stack frame

- Deallocation timing; allow "jump" back in recursive stack

Stack frame: return address of the caller, parameters for the function, storage for local variables, etc.
Stack pointer: top/bottom stack region; first unused location



Frame pointer: fixed location in a stack frame; displace to access elements
Function call convention: non-universal ways to setup stack frame
CS2106 convention:

Caller:	Push $fp and $sp to stack
	Copy $sp to $fp
	Reserve space on stack for parameters by moving $sp
	Write parameters to stack using offsets of $fp
	JAL to callee
Callee:	Push $ra to stack
	Push current register values used in function
	Use $fp to access parameters
	Compute results
	Write result to stack (where $fp is)
	Restore registers saved
	Get $ra from stack
	Return to caller using JR $ra
Caller:	Get result from stack
	Restore $sp and $fp

Register spilling: move register value to memory temporarily when no registers available

Dynamically allocated memory

- Memory allocated during execution time using malloc()
- Size unknown during compilation time, no definite deallocation timing

Heap memory: store information for dynamically allocated values

- Trickier to manage because of variable size and allocation/deallocation timings

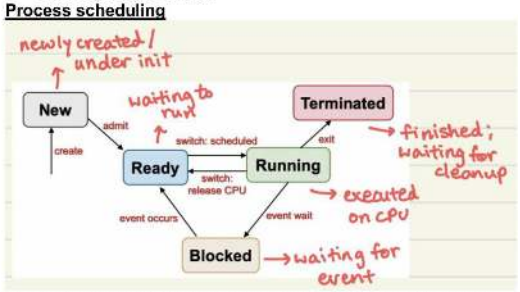
Process

- Dynamic abstraction & information for executing program

Cores <> Processes: m CPUs has at most m processes
init: root process for all processes (PID = 1)

- Created by kernel at boot up time
- Like Supervisor in Elixir

Process scheduling

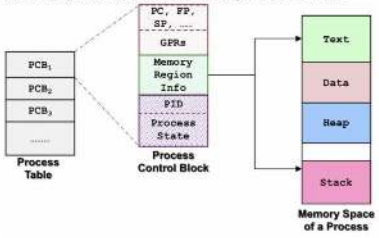


Zombie: child process terminates but parent does not call wait()
Orphan: parent process terminates before child process

- init process becomes pseudo-parent (handles termination)

Process Control Block (PCB)

- Entire execution context for a process
- Maintained by kernel; stored under **process table**



System calls

- Must change from user mode -> kernel mode

Function Wrapper	Function Adapter
Syscall with library version with same name and parameters	User friendly syscall with less params/more flexible param

Mechanism:

1. User program invokes library call
2. Library call places system call number in a designated location like a register
3. Library call executes special instruction to switch from user to kernel mode (TRAP), saving the CPU state
4. Appropriate system call handler is determined by system call number (index), handled by dispatcher
5. System call handler executed
6. System call handler ends, restoring CPU state and returning to library call, switch back to user mode
7. Library call return to user program

Exception & Interruption

Exception: machine level instructions (checked exceptions)

- Synchronous and must be executed by an exception handler

Interrupt: external events, hardware related (unchecked exception)

- Asynchronous and suspends program execution, must be executed by interrupt handler

Handler: could do nothing

1. Save register/CPU state
2. Perform handler routine
3. Restore register/CPU
4. Return from disruption

fork()

```
int fork();
```

- Returns PID of child process (in parent) OR 0 (in child)
- Creates new process, duplicates the current executable machine; executes the remaining code below the fork()
- Memory in child is copied from parent (copy-on-write)

Implementation:

1. Create address space of child process (virtual address space)
2. Allocate p' = new PID
3. Create kernel process data structures (process table entry)
4. Copy kernel environment of parent process (priority)
5. Initialize child process context: PID = p', PPID = parent PID, 0 CPU time
6. Copy memory regions from parent (expensive, optimised with copy-on-write)
7. Acquires shared resources (open files, pwd)
8. Initialize hardware context for child process (copy registers from parent process)
9. Ready to run (add to scheduler queue)

Copy on Write: optimization for memory copying operation where memory locations that are written to are duplicated; noop if reading only

execl()

```
int execl(const char *path, const *char arg0, ..., const *char argN, NULL);
```

- Replaces code but not process information (same PID)
- fork() + execl() on child process to separately run tasks

exit()

```
void exit(int status);
```

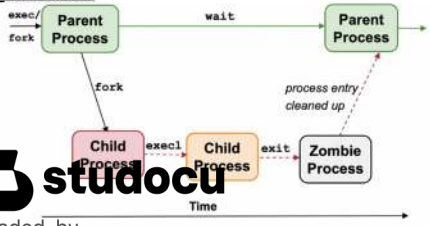
- Ends execution of process
- Status 0 success, !0 error
- Most system resources released like file descriptors
  - Left with PID & status, PCB
- Implicitly done when returning from the main() function

wait()

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

- Parent waits for immediate child process (only) to terminate
- Returns PID of terminated child process
- Status is passed by address (wait(&status)); NULL if unused
- Blocking behavior on parent until one child terminates
- Clears remainder child sys resources, kills zombie processes
- execl() does not affect wait()
- If not children, terminate

Process creation



Concurrent Execution

- Multiple processes progress in execution at the same time
  - Virtual (illusion) or physical (multiple CPUs)

Time slicing: interleaving instructions for multiple processes

- Context switching between processes incur overhead

Process Environments

Batch processing: No user interaction, no need to be interactive	**Turnaround time:** finish – arrival **Waiting time:** time spent in queue **Throughput:** number of tasks finished per unit time **CPU utilization:** % time where CPU is not idling
Interactive: Active users interacting with system, must be responsive	**Response time:** first CPU use – arrival time **Predictability:** variation in response time
Real-time processing: Deadline to meet, usually periodic process	

Scheduling Algorithm Criteria

1. Process Environment
2. Fairness: all process gets fair share of CPU; no starvation
3. Utilization: all parts of system should be used (CPU, I/O)

Types of Scheduling

1. Non-preemptive (cooperative): process keeps running until blocked or gives up CPU voluntarily
2. Preemptive: process given fixed time quantum to run
   - Can be blocked or give up early
   - Forced to suspend once time quantum up
   - Ensures good response time as scheduler runs periodically
   - Periodic timer interrupt that cannot be intercepted; invokes scheduler
   - **Interval of Timer Interrupt (ITI):** OS scheduler invoked every timer interrupt (usually 1-10ms)
   - **Time quantum:** execution duration given to a process; in multiples of ITI; can be variable (usually 5-100ms)

Scheduling Steps

1. Scheduler is triggered (OS takes over)
2. If context switch is needed, context of current running process is saved and placed on blocked queue/ready queue
3. Pick a suitable process P to run based on scheduling algorithm
4. Setup context for P
5. Let P run

Scheduling Algorithms

FCFS	*Batch processing, non-preemptive* Tasks stored in queue based on arrival time & tasks run till done or blocked Blocked tasks are removed from queue and when I/O completes, added back to queue Guarantees no starvation as every task runs Average waiting time can be improved and convoy effect present where long CPU bound task blocks I/O and vice versa Shortest average response time if jobs arrive in order of increasing lengths or all jobs have same completion time
SJF	*Batch processing, non-preemptive* Select task with smallest total CPU time (needs this info in advance/guess) Prediction algorithm from previous CPU-bound phases using exponential average $Predicted_{n+1} = \alpha Actual_n + (1-\alpha) Predicted_n$ Minimizes average waiting time given a fixed set of tasks Starvation is possible due to bias towards short jobs
SRT	*Batch processing, preemptive* Variation of SJF using remaining time New jobs with shorter remaining time pre-empts currently running jobs; good for short jobs with late arrival

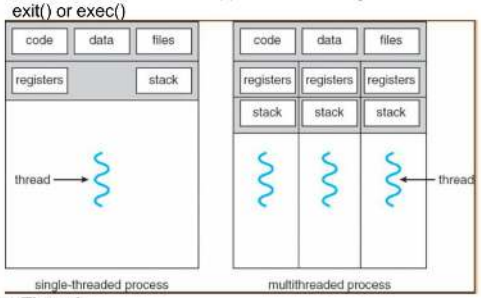
This document is available on studocu
messages.downloaded_by

RR	*Interactive, preemptive* Like FCFS but preemptive so when time quantum lapses, task forced to give up CPU and placed to end of queue (after I/O) Response time (guaranteed) to be upper bounded by (n - 1)q with n tasks and time quantum q Larger time quantum yields better CPU utilization but longer waiting time, shorter time quantum yields larger context switching overhead but shorter waiting time Behaves like FCFS if job lengths are shorter than time quantum Performs worse than FCFS if job lengths are all the same and greater than time quantum (higher average turnaround time) or when there are many jobs and job lengths exceed time quantum (reduced throughput due to increased overhead)
Priority Scheduling	*Interactive, preemptive/non-preemptive* Assigns priority to processes and run highest priority first Preemptive: Higher priority process can preempt running process with lower priority Non-preemptive: Late coming waits for next round of scheduling Low priority process can starve if high priority keeps hogging CPU (worse for preemptive); resolved by decreasing priority or current process after each time quantum or giving current process a time quantum **Priority inversion:** lower priority preempts higher priority because resource is locked so higher priority is blocked
MLFQ	*Interactive, preemptive* Multiple queues with different priority levels, minimizes response time for I/O bound processes and turnaround time for CPU bound processes Highest priority queue must be empty before next queue is used Rules: 1. Priority(A) > Priority(B) -> A runs 2. Priority(A) == Priority(B) -> A and B runs in RR 3. New job -> highest priority 4. Job fully used time quantum -> reduce priority 5. Job gives up before time quantum -> retain priority **Change of heart:** process with lengthy CPU phase right before I/O phase sinks to the lowest priority and gets starved; periodically move all tasks to highest priority to fix **Gaming the system:** process repeatedly gives up CPU before time quantum lapses remains in high priority and starves other processes; accumulate total CPU use time across all quantums
Lottery Scheduling	*Interactive, preemptive* Give out "lottery tickets" to processes for system resources, randomly choose lottery ticket, winner gains resource; lottery tickets can be distributed to children Process holding X% of tickets has X% chance to win and use X% of resource Responsive as new processes can participate

Threading
- Units of work within a process
- Shares memory context (text, data, heap) and OS context (process ID, files, etc.)
- Uniquely identified by thread ID, registers, and stack

Thread switching: change hardware context like registers and stack
Benefits:
1. Economy: multiple threads in the same process requires less resources to manage
2. Resource sharing: threads share most of the resources of a process
3. Responsiveness
4. Scalability: take advantage of multiple CPUs

Problems:
1. System call concurrency: ensure that parallel system calls have no race condition
2. Process behavior: what happens when a single thread calls exit() or exec()



User Thread
- Thread implemented as user library (process handles thread related operations)
- Kernel not aware of user threads
- User thread with no kernel thread still runs on a single thread

Pros:
1. Can have multithreaded program on any OS
2. Thread operations are just library calls
3. More configurable and flexible

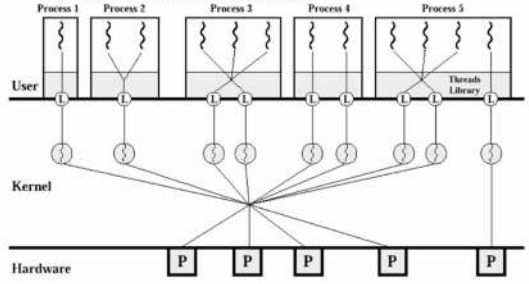
Cons:
1. OS not aware of threads and scheduling is performed at process level
   - If one user thread blocks, the entire process is blocked so all threads are blocked
   - Cannot exploit multiple CPUs

Kernel Thread
- Thread is implemented in the OS handled as system calls
- Kernel schedule by threads, not processes; may make use of threads for its own execution

Pros:
1. Kernel schedules on thread levels
   - Multiple threads can run for the same process and be non-blocking

Cons:
1. Thread operations are system calls that are slower and more resource intensive
2. Less flexible since kernel threads used by all multithreading programs

Hybrid Thread
- User threads bind to a kernel thread



Simultaneous Multi-threading
- Supply multiple sets of registers to allow threads to run natively and in parallel on the same core

POSIX thread
- Implemented as either user or kernel threads
- **pthread_t**: data type to represent thread id
- **pthread_attr:** data type to represent attributes of thread

pthread_create:

```
int pthread_create(
    pthread_t* tidCreated,
    const pthread_attr_t* threadAttributes,
    void* (*startRoutine) (void*),
    void* argForStartRoutine);
```

Return 0 if successful, !0 if errors

tidCreated: thread id for the created thread
threadAttributes: control the behavior of the new thread
startRoutine: function pointer to the function to be executed by thread
argForStartRoutine: arguments for the startRoutine function

pthread_exit: pthread terminates automatically at the end of the startRoutine with return value defined by return statement

```
int pthread_exit(void* exitValue);
```

exitValue: value to be returned to whoever synchronize with this thread

pthread_join: wait for the termination of another pthread

```
int pthread_join(pthread_t threadID, void **status);
```

Return 0 if successful, !0 if errors

threadID: TID of the pthread to wait for
status: exit value returned by the target pthread

Shared Memory (IPC)
- Shared between processes
- Process P1 creates shared memory M (shmget) and P2 attaches M to its memory space (shmat)
- OS involved in creating and attaching memory region; easy to use as all operations are array operations
- Requires synchronization and implementation is harder
- Must detach M (shmdt) and destroy M (shmctl) after all processes detached

Message Passing (IPC)
- Process P1 sends message M to P2 and P2 receives M (all as system calls)
- Additional properties: identification of other party (naming scheme) and synchronization of send/receive operations
- Messages are stored in kernel memory space

Direct communication: sender/receiver of message explicitly name the other party
- One link per pair of communicating processes
- Identify of other party must be known

Indirect communication: messages are sent to/received from message storage (mailbox/port)
- One mailbox shared among multiple processes

Synchronization behaviors for send() and receive(): blocking primitives (synchronous) or non-blocking primitives (asynchronous)
Advantages: easier to implement on different processing environment and easier to synchronize
Disadvantages: requires OS intervention so inefficient and harder to use due to limits on message size/format

Unix Pipes (IPC)
- Share input/output between processes (producer/consumer)

Process communication channels: stdin, stderr, stdout
Piping (A | B): links the input/output channels of one process to another
Pipe in C: circular bounded byte buffer with implicit synchronization (writer waits when buffer is full, reader waits when buffer is empty)
Unidirectional (half-duplex): one write end, one read end
Bidirectional (full-duplex): any end for read/write

System calls:

```
int pipe(int fd[])
close(int) -> side to close
write(int, data, length) -> side to write to
read(int, buffer, length) -> read from
```

0 for success, !0 for errors

fd[0] is reading, fd[1] is writing

- Must close the end not in use, otherwise, other processes might accidentally write/read to it

Redirecting communication channels to pipes: use dup() and dup2()
- Sets new file descriptor to refer to old as well (i.e. old and new are the same now)

dup2(fd[0], STDIN_FILENO)	Redirect input to read from read end of pipe
dup2(fd[1], STDOUT_FILENO)	Redirect standard output to write to write end of pipe
dup2(file, STDOUT_FILENO)	Redirect standard output to write to contents of file

Unix Signals (IPC)
- Asynchronous notification regarding an event sent to a process/thread
  - E.g. kill, stop, continue
- Recipient of the signal must handle the signal via default handlers or user supplied handlers (signal(type, handler))

Race Conditions
- Execution outcome depends on the order in which the shared resource is accessed/modified (non-deterministic)
- General modification flow: load memory value into register, update register value, load register to memory
  - Race conditions happen when loading happens at the wrong time
- Caused by unsynchronized access to shared modifiable resources

Solution: designate code segment with race condition as critical section, only one process can execute in CS (synchronization)

Critical Section

```
// Normal code
Enter CS
// Critical work
Exit CS
// Normal code
```

Mutual exclusion: if process is executing in critical section, all other processes are prevented from entering the section
Progress: if no process is in a critical section, one of the waiting processes should be granted access
Bounded wait: after a process requests to enter the critical section, there exists an upper bound on how many other processes can enter the section before the process
Independence: process not executing in critical section should never block other process

Incorrect Synchronization
1. Deadlock: all processes blocked so no progress
2. Livelock: related to deadlock avoidance mechanism where processes keep changing state to avoid deadlock and make no other progress; processes are not blocked
3. Starvation: some processes never get to make progress in their execution as it is perpetually denied necessary resources

Test and Set

TestAndSet(Register, MemoryLocation)
- Machine instruction provided by processors to aid synchronization
- Load the current content at MemoryLocation into Register and stores 1 into MemoryLocation (treated as lock)
- Performed as single machine operation (atomic)
- Used to create spinlocks

```
void EnterCS(int* Lock) {
    // Cannot enter if lock value is 1
    while (TestAndSet(Lock) == 1);
}
```

```
void ExitCS(int* Lock) {
    *Lock = 0;
}
```

Busy waiting: keep checking the condition until it is safe to enter critical section which is wasteful use of processing power
Variants: compare and exchange, atomic swap, load link/store conditional

messages.downloaded_by