CS2106

OS

- Is a program that acts as an intermediary between user and computer hardware
- Manages resources and coordinates requests (process synchronization, resource sharing)
- Simplifies programming (abstraction of hardware, convenient services)
- Enforces usage policies
- Provides security and protection (as a control program)
- User program portability (across different hardware)
- Efficiency (optimized for particular usage and hardware)

Kernel mode Have complete access to all hardware resources

User mode With limited (or controlled) access to hardware resources

OS types

Monolithic Kernel is one big special program (e.g. most Unix variants, Windows NT/XP)

✓ Well understood; Good performance

× Highly coupled components; Usually very complicated internal structure

Microkernel Kernel is small, providing only basic essential facilities (e.g. IPC, address space & thread management)

Higher-level services (e.g. device driver, process & memory management, file system) run as server process outside of OS, using IPC to communicate.

✓ Kernel is more robust & extendible; Better isolation and protection between kernel and high-level services

× Lower performance

- Address space refers to the set of memory locations that a program has access to

Layered systems Generalization of monolithic, where components are organized into layers, which each serve a specific role

Client-Server Variation of microkernel; Client and server can be on separate machines

VMs

Purpose

- Run multiple OS on same hardware
- Observe inner working
- Test potentially destructive implementation

Type 1 Hypervisor

- Runs directly on hardware (e.g. IBM VM/370)
- Harder to implement but better performance

Type 2 Hypervisor

- Runs on host OS (e.g. VMware);
- Easier to implement but worse performance
- Independent of actual hardware, can use to emulate hardware you don't have

Process Abstraction

Process An abstraction to describe a running program

Memory context A process has:

- Text: for instructions
- Data: for global variables
- Heap: for dynamic allocation
- Stack: for function invocations

Note that pointers are stored as storing raw data is impractical (but OS must ensure that the data of switched-out process is not modified)

Hardware context

- General purpose registers (GPRs)
- PC, SP, FP, etc.

Register spilling If GPRs are insufficient, use memory to temporarily hold GPR values

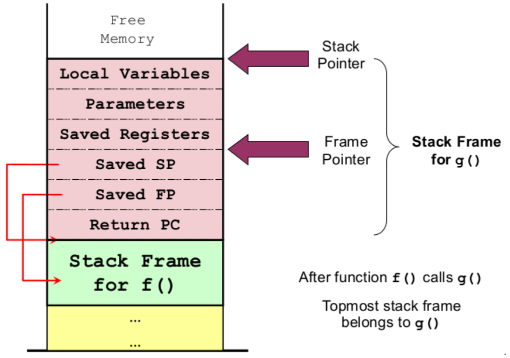
Stack frame example

Executing function call

1. Pass arguments with registers and/or stack
2. Save return PC on stack
3. **Transfer control from caller to callee**
4. Save registers to be used by callee. Save old FP, SP
5. Allocate space for local vars of callee on stack
6. Adjust SP to point to new stack top

Returning from function call

1. Restore saved registers, FP, SP
2. **Transfer control from callee to caller using saved PC**
3. Continues execution in caller

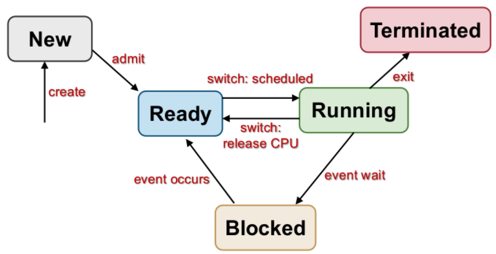


Misc

- Saved SP is important in architectures without `PUSH` and `POP` instructions (where you need to load and store to address pointed at by SP)
- SP may change, FP stays constant
- Stack frame sizes may not be computable at compile time, if we use `alloca` (not required for CS2106)
- There are scenarios where SP, FP are not required
- Stack teardown is achieved by modifying SP. Results stay on stack until overwritten

Process ID & State

Generic 5-state process model



Process control block (PCB) Stores the entire execution context for a process:

- Memory, hardware contexts
- OS context: PID, process state

System calls

- Implemented as a kernel-mode routine with some parameters

In C/C++ Can be almost directly invoked

- Most have a library version with same name and params, acting as a function wrapper
- Few have a more user-friendly version with flexible params, acting as a function adapter

General mechanism

1. User program invokes library call
2. Library call places syscall number in designated location
3. Library call executes TRAP instruction to **switch from user mode to kernel mode**
4. Appropriate syscall handler is determined, using syscall number as index (usually handled by a dispatcher)
5. Syscall handler is executed (carries out actual request)
6. Return control to library call, switch from kernel mode to user mode
7. Library call returns to user program

Exception/Interrupt/Signal

Exception Synchronous (occurs due to program execution)

- Executes an exception handler, like a forced function call
- Traps are intentionally set up exceptions

Interrupt Asynchronous (occurs independent of program execution, usually hardware related)

- Executes an interrupt handler, program execution is suspended
- From hardware to OS kernel

Signal High level communication mechanism in Unix OSes

- Asynchronous wrt program that receives them
- When an exception occurs, info may be delivered to a process via a signal

Process Abstraction in Unix

Process information

- PID (integer)
- Process State (running, sleeping, stopped, zombie)
- Parent PID
- Cumulative CPU time, etc.

C command line args

- `main` has signature `int main(int argc, char* argv[])`
- `int argc` contains the number of command line args (including program name)
- `char* argv[]` contains the command line args as C character strings

Create `int fork()`

- `#include <unistd.h>`
- Returns PID of newly created process (to parent process), and 0 (to child process)
- Child is basically a duplicate of parent, with the execption of PID and PPID
- Memory regions are copied from parent
- Child starts running from the immediate **machine instruction** after the fork
- Linux provides `clone()`, a more verstaile version of `fork()`, that allows specification of what execution context can be shared
- Running a command in a terminal is essentially a `fork()` followed by `exec()`

Fork implementation

1. Create address space of child
2. Allocate new PID, p'
3. Create kernel process data structures (e.g. entry in process table)
4. Copy kernel environment of parent process (e.g. priority)
5. Initialize child process context (PID=p', PPID=parent id, CPUtime=0)
6. Copy memory regions from parent (expensive, can be optimized, next section)
7. Acquires shared resources (open files, cwd)
8. Initialize hardware context for child (copy registers, etc.)
9. Add child to scheduler queue

Copy on write

- Child potentially needs to copy the whole memory space
- Child might not need the entire memory space immediately
- If child just reads, then can use shared version
- Only if ANY write occurs, then must have independent copies of data

Execute `exec` family of calls

- `#include <unistd.h>`
- Replaces current executing process image with new one
- Only code is replaced, but PID and other information stays intact
- Has several different variants: `execl, execv, execve, excle, execvp`
- `int execl( const char *path, const char *arg0 ... *argN, NULL )`
- e.g. `execl("/bin/ls", "ls", "-l", NULL)`, same as executing `ls -l` in terminal

Terminate `void exit(int status)`

- Status is returned to the parent process
- 0 indicates normal termination, not 0 indicates problematic execution
- Does not return (control flow does not go to callee), but exit status is communicated to callee
- Most programs have no explicit `exit()` call, but returning from `main()` implicitly calls `exit()`.

Master process `init`

- Created in kernel at boot up time
- Traditionally has PID = 1
- Root process

On exit

- Most system resources are released (e.g. file descriptors)
- Some basic resources are not releasable (e.g. PID, status - for parent-child sync; CPU time - for process accounting)
- Process table entry may still be needed

Wait `int wait(int *status)`

- Returns PID of terminated child process
- `int *status` stores the exit status of the terminated child. Init to NULL if this info is not needed.
- Blocking: parent blocks until one child terminates (then removes child from process table)
  - Even if child is a new executable (`exec`), still waits
  - If there are no children, does not block
- Cleans up remainder of child system resources not removed in exit
- Variants: `waitpid()` that waits for a specific child process, `waitid()` that waits for any child process to change status

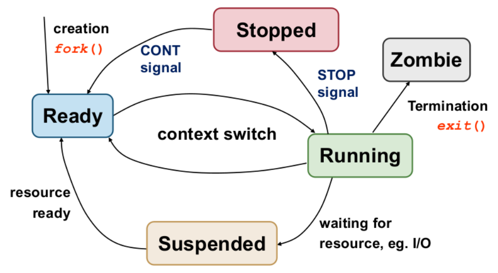
Zombie process

- `wait()` creates zombie processes, as the (dead) child process might need to pass information to the parent
- If parent terminates first, then `init()` becomes the pseudo parent. Child termination sends a signal to `init()` which uses `wait()` to cleanup
- If child terminates first (and parent did not `wait()`), child becomes zombie, which can fill up the process table.
- On older Unix implementations, may need a reboot to clear the table
- If parent never terminates, zombie processes will continue to exist

Get PID

- `pid_t` is a signed integer type, but in GNU this is an `int` instead.
- `pid_t getpid()` returns the process PID
- `pid_t getppid()` returns the parent process PID

Unix process model



Process scheduling

- When does the OS/scheduler run?
  - Woken up by events (e.g. data arrival, timer interrupt, process termination)
  - Save context of running process
  - Do what OS needs to do
  - Restore context
- I/O signals are handled by interrupt service routines (not user-space processes)
- If there is only 1 process, the scheduler let it run without intervening (save on overhead and context switching)

Non-preemptive

- Process stays scheduled until it blocks, or gives up CPU voluntarily
- Lesser overhead

Preemptive

- At the end of the time quota, the process is suspended (a process might still block or finish/give up CPU early)
- Used when responsiveness of system is important

CPU/Compute-bound Process spends most of its time using CPU

IO-bound Process spends most of its time using I/O

Batch processing

No user interaction, non-preemptive scheduling is predominant

Criteria

- Throughput: Number of tasks finished per unit time
- Turnaround time: Total wall clock time (finish - start), includes waiting time
- CPU utilization: Percentage of time when CPU is working

First come first served (FCFS)

- Maintain a FIFO queue based on arrival time
- Blocked task is removed from queue, and placed at back once it is ready
- Non-preemptive
- No starvation: number of tasks before X in FIFO is always decreasing
- Simple reordering can reduce average waiting time (cf. SJF)

Convoy effect

- CPU-bound task A is followed by many IO-bound tasks $X_1, X_2, \cdots, X_N$
- Task A runs, while tasks X_i wait in ready queue (I/O is idle)
- Task A blocked on I/O, so tasks X_i execute on CPU quickly, now blocked on I/O (CPU is idle)

Shortest Job First (SJF)

- Select task that takes shortest amount of CPU time
- Need to know total CPU time for a task in advance. Common approach is exponential average:

$$\text{Predict}_{t+1} = \alpha \text{Actual}_t + (1-\alpha)\text{Predict}_t$$

- α represents the significance of the immediate past value
- $1-\alpha$ represents the significance of the past history

- Can be preemptive or non-preemptive
- Starvation is possible; biased towards short jobs

Shortest Remaining Time (SRT)

- Preemptive version of SJF, using remaining CPU time
- New job with shorter remaining time can preempt currently running job
- If all tasks arrive at beginning, SRT gives same schedule as SJF
- Starvation is possible

Interactive systems

Criteria

- Response time: Time between request and response by system
- Predictability: Variation in response time (important in real-time environments)
- Interval of timer interrupt (ITI): OS scheduler is triggered every ITI (typically 1ms to 10ms)
- Time quantum: Execution duration given to a process, must be a multiple of ITI (typically 5ms to 100ms)

Round robin (RR)

- Preemptive version of FCFS, where task gets interrupted after time quantum elapses
- Given n tasks and quantum q, the time before a task gets CPU is upper bounded by $(n-1)q$, i.e. starvation not possible
- Choice of time quantum is important
  - Big: better CPU utilization; longer waiting time
  - Small: worse CPU utilization (bigger overhead); shorter waiting time
- Same as FCFS if all jobs are shorter than TQ

Priority scheduling

- Each task gets a priority, and highest priority gets scheduled first
- Preemptive version: Higher priority process can preempt
- Non-preemptive version: Wait for next round of scheduling

Priority scheduling (starvation)

- Starvation is possible: low priority process
- Possible solution: decrease priority of current process after every time quantum
- Possible solution: give each process a minimum time quantum (so they don't get preempted)

Priority scheduling (priority inversion)

- LP process locks resource, MP process preempts (and resource is still locked), HP process requiring resource is starved by MP process
- Lower priority task effectively preempts higher priority task
- Solution: (Priority inheritance) Temporarily increase priority of LP to HP, until it unlocks the lock, then restore original priority

Multi-Level Feedback Queue (MLFQ)

- If $Priority(A) > Priority(B)$, A runs
- If $Priority(A) == Priority(B)$, A and B run in RR
- New job gets highest priority
- If a job fully utilize TQ, priority reduced
- If a job gives up/blocks before fully utilizing TQ, priority retained
- Starvation is possible

MLFQ problems

1. Process with lengthy CPU-intensive phase followed by IO-intensive phase
   - Process may sink to lowest priority during CPU-intensive phase
   - Fix: periodically reset all processes to highest priority (treat all as new)
2. Process repeatedly gives up CPU just before TQ lapses
   - Process is able to retain its high priority, tricking the system, receiving disproportionate amount of CPU time
   - Fix: track accumulative CPU time instead of counting from 0. Will reach TQ (and priority will be lowered)

Lottery scheduling

- Scheduling done in rounds, where each process gets ≥ 1 tickets.
- When scheduling decision is required, a ticket is drawn without replacement
- In worst case, max response time is 2 scheduling rounds (1 round if just missed ticket distribution, another if it is unlucky and runs last in the round), thus starvation not possible
- Each resource can have its own set of tickets

Inter-Process Communication

Shared memory

- Communication through reads/writes to shared variables

Advantages

- Efficient: only the initial setup requires OS
- Ease of use: shared memory is just like a normal memory space

Disadvantages

- Limited to a single machine (an abstraction of shared memory may work in distributed systems, but less efficiently)
- Requires synchronization to prevent race conditions

POSIX shared memory in *nix

Usage Note: OS only involved in first 2 steps (not just in *nix)

1. Create/locate a shared memory region M `int shmget(key_t key /*IPC_PRIVATE*/, size_t size, int shmflg /*IPC_CREAT | 0600*/)`
2. Attach M to process memory space `void *shmat(int shmid, const void *shmaddr /*NULL*/, int shmflg /*0*/)`
3. Read from/Write to M
4. Detach M from memory space `int shmdt(const void *shmaddr)`
5. Destroy M (only one process does this, and can only destroy if M is not attached) `int shmctl(int shmid, int cmd /*IPC_RMID*/, struct shmid_ds *buf /*08/)`

Message passing

- Message stored in kernel memory space
- All send/receive go through OS (i.e. syscall)

Direct communication

- Sender/receiver explicitly names other party
- One link per pair of communicating processes

Indirect communication

- Sender/receiver uses one mailbox
- Can be shared among multiple processes
- Usually used when it doesn't matter who received what message
- Programmer's responsibility that the processes use the correct mailbox

Blocking primitives (synchronous)

- `send()` blocks until message received
- `receive()` blocks until message arrived
- No intermediate buffering required
- Easier to reason about

Non-blocking primitives (asynchronous)

- `send()`
  - sender resumes operation immediately
  - BUT because buffer required, if buffer is full, sender might wait (becomes synchronous) or return with error
- `receive()`: if message has not arrived, proceeds and does not block
- Better performance but need more careful programming

Advantages

- Portable (implement on different platforms)
- Easier synchronization (blocking semantics of send/receive implicit synchronize sender/receiver)

Disadvantages

- Inefficient (needs OS intervention for every send/receive)
  - In shared memory, OS is only involved in setup
- Harder to use (message must follow supported message format)
  - In shared memory, data can have any form

Unix pipes

```
#include <unistd.h>
int pipe(int fd[2]); // create new pipe
fd[0], fd[1] // file descriptors for
    reading, writing
```

- In Unix, a process has 3 default communication channels: stdin (0), stdout (1), stderr (2)
- Functions as a circular bounded byte buffer with implicit synchronization
  - Writers wait when buffer is full
  - Readers wait when buffer is empty
- (Variant) can have multiple readers/writers
- (Variant) may be unidirectional or bidirectional

Unix signals

```
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum,
    sighandler_t handler);
// returns previous signal handler, or
    SIG_ERR on error
```

- Asynchronous notification regarding an event, sent to a process/thread
- Must handle the signal, either using default handler, or user supplied handler (only some signals)

Threads

Motivation

- Process is expensive (process creation: duplicate context, context switch: save/restore process info)
- Communication between processess is challenging and inefficient (requires IPC or shared memory regions)

Description

- A single process can have multiple threads
- Unique info for each thread (ID, registers, "Stack": just changing FP and SP registers)
- Threads in same share memory, OS contexts
- Thus, thread context switch only involves hardware context (and not memory, OS contexts)

Benefits

- Less resources needed (cf. processes)
- No need additional mechanism for passing info
- Multithreaded programs can appear more responsive
- Multithreaded program can take advantage of multiple CPUs

Problems

- Synchronization around shared memory gets even worse (since all memory except stack is shared between threads)
- Parallel/concurrent syscalls possible (OS must guaranatee correctness)
- Process behaviour (how does `fork()`, `exit()`, `exec()` work with multiple threads
- Less security between threads, compared to between processes

User threads

- Threads are implemented as a user library
- Kernel is not aware of threads within process

Advantages

- Can be used on any OS
- Thread operations are just library calls
- Generally more configurable, flexible (e.g. user-customized thread scheduling policy)

Disadvantages

- OS is not aware of threads, so scheduling is performed at process level
- One thread block ⇒ Process blocked ⇒ All threads blocked

Kernel threads

- Threads are implemented in OS
- Thread-level scheduling is possible
- Kernel often uses threads for its own execution

Advantages

- Threads from same process can be run simultaneously on multiple CPUs
- If one thread is blocked, other threads can still continue

Disadvantages

- Thread operations are syscalls (slower, more resource intensive)
- Generally less flexible (as it needs to support all multithreaded programs)

Hybrid thread model

- User thread binds to a kernel thread
- Can limit concurrency of any process/user

Modern processors

- Has hardware support (multiple sets of registers)
- Allows threads to run natively in parallel on the same core (known as simultaneous multithreading)

Note

- Can specify a function you want a pthread to run when you spawn it

Synchronization

Race conditions

- Execution of concurrent processes may be non-deterministic
- Outcome depends on order in which shared resource is accessed/modified
- Need synchronization to control the interleaving of accesses to a shared resource

Critical section (CS)

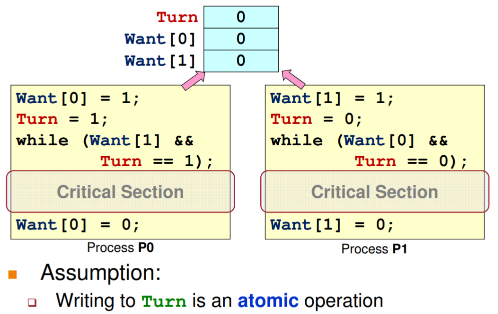
Properties of correct implementation

- Mutual exclusion: If process in CS, then all other processes cannot enter CS
- Progress: If no process in CS, then one waiting process should be granted access
- Bounded wait: After a progress requests access to CS, there exists an upper bound on the number of times other processes can enter CS before this process
- Independence: Process not executing in CS should never block other processes
- Note: progress ⇒ independence, but independence ⇏ progress

Symptoms of incorrect synchronization

- Incorrect behaviour: Usually due to lack of mutual exclusion
- Deadlock: All processes blocked ⇒ no progress
- Livelock: Processes are typically not blocked, but they keep changing state to avoid deadlock, and make no other progress
- Starvation: Some processes are blocked forever

Peterson's algorithm



Disadvantages

- Busy waiting: Using CPU cycles to wait for something to happen
- Low level implementation
- Not general

Test and Set

- `TestAndSet Register, MemoryLocation`
- Single atomic machine operation, even in multicore systems
- Loads current content at MemoryLocation to Register, then stores a 1 into MemoryLocation
- `while(TestAndSet(MemoryLocation)== 1);`
- Only if you locked it, you will get 0

Semaphores

`Wait(S)`

- If S ≤ 0, blocks
- Decrement S
- Also known as `P()` or `Down()`

`Signal(S)`

- Increment S
- Wakes up one sleeping process (if any)
- Never blocks
- Also known as `V()` or `Up()`

Invariant Given $S_{\text{initial}} \geq 0$,

$$S_{\text{current}} = S_{\text{initial}} + N_{\text{signal}} + N_{\text{wait}}$$

- N_{signal}: number of `signal()` operations executed
- N_{wait}: number of `wait()` operations completed

Variants

- Binary semaphore: $S = 0$ or 1
- General (counting) semaphore: $S \geq 0$
- General semaphore can be mimicked by binary semaphores

Mutex

- Implemented as a binary semaphore
- Invariant:

$$S_{\text{current}} + N_{\text{CS}} = 1$$

Misc

- Semaphore can be used to block process B until process A finishes executing a particular statement
- No unknown unsolvable synchronization problem with semaphore
- Alternative: conditional variable
- Always consider both single-core and multi-core systems