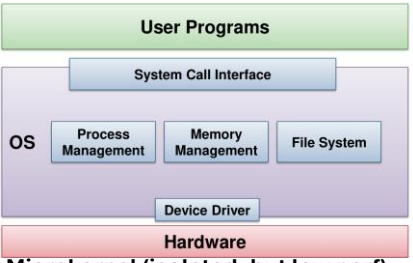
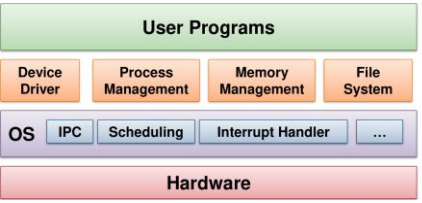
Monolithic Architecture

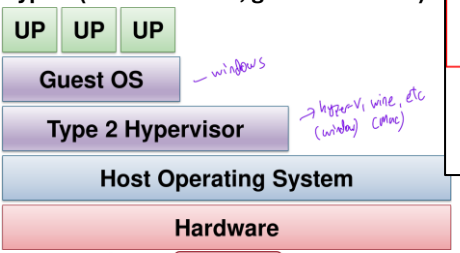


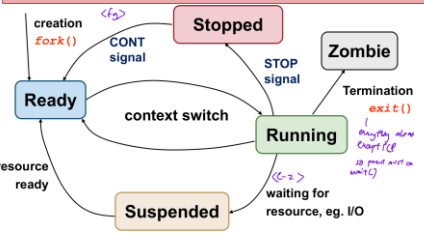
Microkernel (isolated, but low perf)
Provides IPC, Addr space/Thread mgm
High Level Services
run as server process outside OS, communicate using IPC, Better isolation between kernel and HLS



Virtual Machine / Hypervisor
Virtualization of underlying hardware
Type 1 (individual VM to guest OS)

Type 2 (run in host OS, guest run in VM)

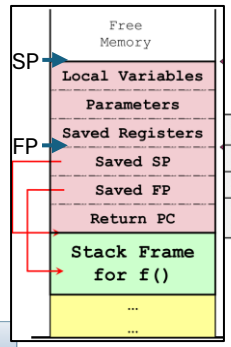




Instruction Execution
Memory indicated by PC Instruction, read from mem or GPR
Memory Context:
- text (instructions),
- data (global vars)
- Stack, Heap
Hardware Context:
GPR, PC, SP, FP
OS Context:
PID, Process State
Stack Memory
Stores function invoc in a **stack frame**
Contains:
- Return addr of caller
- function args
- local vars

Saved Registers
(register spilling)
- Use memory to hold GPR
- GPR can be used for other purposes and restored later

Stack Frame

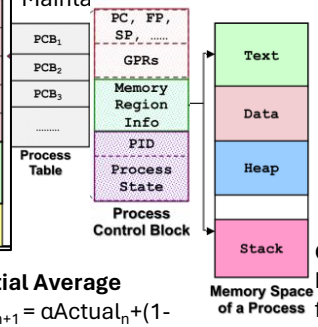


Stack Frame Setup/Teardown
<u>On executing function call:</u>
- Caller: Pass arguments with registers and/or stack
- Caller: Save Return PC on stack

<u>Transfer control from caller to callee</u>
- Callee: Save registers used by callee. Save old FP, SP
- Callee: Allocate space for local variables of callee on stack
- Callee: Adjust SP to point to new stack top

<u>On returning from function call:</u>
- Callee: Restore saved registers, FP, SP
- Transfer control from callee to caller using saved PC
- Caller: Continues execution in caller

Dynamic Memory (Heap)
Runtime known only at compile time (not data), no definite allocation/dealloc timing (not stack)
PCB
Maintained by Kernel



Exponential Average
$Predicted_{n+1} = \alpha Actual_n + (1-\alpha) Predicted_n$
- $Actual_n$ = The most recent CPU time consumed
- $Predicted_n$ = past history
- α = Weight placed on recent event
- $Predicted_{n+1}$ = Latest prediction

Syscalls
1. Library places syscall no. in a register
2. Switch from user to kernel
3. Dispatcher uses syscall no. as index to execute handler
4. Return control to library
5. Switch from kernel to user

Exceptions
- Synchronous
- Execute exception handler

Interrupt
- Asynchronous,
- Usu. hardware related
- Suspend program execution
- Executes interrupt handler

int fork()
Returns child pid for parent,
Returns 0 for child
int main(int argc, char* argv[]);
argc: num of args, inc program name
argv: each element is a C char string
- If 3 params, argc = 4
- arg0 is a.out, arg1-3 are params

int execl(const char *path, const char *arg0, ..., NULL);
execl("/bin/ls", "ls", "-l", NULL);
Path = "/bin/ls", arg0 = "ls", arg1 = "-l"
void exit(int status);
0 = normal, !0 = problem
int wait(int *status);
blocks then returns PID of terminated child
returns immediately if no child

Criteria: Balance, Fairness (starve)
Response Time:
first CPU time – arrival time
Turnaround Time: finish – arrival
Waiting Time: time waiting for CPU
Throughput: Number of tasks finished per unit time
CPU Utilisation: %time CPU is used
Waiting Time: turnaround – CPU work done – IO waiting time

Zombie Process
Child that exited but parent not called wait() by parent
If Parent process terminates before child process:
- init process becomes "pseudo" parent of child processes
- Child termination sends signal to init, which utilizes wait() to cleanup

Child process terminates before parent but parent did not call wait:
- Child becomes zombie
- Can fill up process table
- May need a reboot to clear the table

Processing Environment
1. Batch:
- No interaction required, No need to be responsive
2. Interactive (or Multiprogramming):
- With active user interacting with system
- Should be responsive, consistent in response time
3. Real time processing:
- Have deadline to meet
- Usually periodic process

Lottery
a process holding X% of tickets uses resource X%
Responsive:
- A newly created process can participate in the next lottery
- Provides good level of control:
- A process can be given Y lottery tickets depending on priority, and to be distributed to child,
- Can control the proportion of usage
- Each resource can have its own set of tickets
- Different proportion of usage per resource per task

<u>Batch</u>
FCFS: no starvation
Convoy Effect: all wait CPU, all wait IO, repeat (a lot of idling)
Shortest Job First:
Minimises ave waiting time
Need to know CPU time
Long jobs starve
Uses exponential average
Shortest Remaining Time (Preemptive)
Good service for late short jobs
Only really need to check when new job comes

<u>Interactive</u>
- Interval of Timer Interrupt (ITI), invoke scheduler every x *ms*
- Time Quantum (a multiple of ITI), execution duration

NB Context switching wastes time
Round Robin (RR)
Store in FIFO until TQ, give up or block
Placed to back of queue
Time before a task gets CPU bounded by $(n-1)q$
Big quantum: Better CPU utilization but longer waiting time
Small quantum: Bigger overhead (worse CPU utilization) but shorter waiting time
Priority Scheduling
Preemptive: higher preempts low
Non: latecomer waits for next round
Low priority can starve
Priority Inversion: Resource locked, lower priority preempts higher priority
Multilevel Feedback Queue (MLFQ)
- Response time for IO bound processes minimised
- Turnaround time for CPU bound processes minimised
- If P(A)>P(B), A runs
- If P(A)==P(B), A and B runs in RR
- Priority reduced if fully utilizes time slice, else retain Priority

Multithreaded processes
- Shared Memory Context: Text, Data, Heap
- Shared OS Context: Process id, stack
Unique information needed for each thread
- thread id
- Registers (GPR, special)
Process Context Switch:
OS, hardware, memory context
Process Thread switch:
Hardware context: registers, FP, SP
Benefits
Share resources
Less kernel mode switches
User Thread:
- Thread is implemented as a user library
- A runtime system (in the process) will handle thread
related operation
- Kernel is not aware of the threads in the process – 1 kernel thread to 1 process entrypoint even if more than 1 process thread
Kernel Thread:
- Thread implemented in OS
- Thread operation is handled as system calls
- Thread-level scheduling is possible:
- Kernel schedule by threads, instead of by process
- Kernel may make use of threads for its own
execution
User Thread Adv and Disadv
+ Multithreaded program on any OS
+ more configurable, flexible
- OS not aware of threads, scheduling performed at process level
- One thread blocked = all threads blocks
- Cannot exploit multiple CPUs
Simultaneous Multithreading (SMT)
Multiple sets of registers to allow threads to run in parallel

Create pthread
```
int pthread_create(
   pthread_t* tidCreated,
   const pthread_attr_t* threadAttributes,
   void* (*startRoutine) (void*),
   void* argForStartRoutine    );
```
returns 0=success, !0=error
Exit pthread
```
int pthread_exit( void* exitValue );
```
if return XYZ instead, then use XYZ instead of *pthread_exit*
Join pthread (synchronise)
```
int pthread_join( pthread_t threadID,
void **status );
```

Shared Memory
Advantages:
- Only create and Attach shared memory region) involves OS
Ease of use:
- Shared memory region behaves the same as normal memory space
- i.e. Information of any type or size can be written easily
Disadvantages:
- Synchronization:
- Shared resource -> Need to synchronize access
- Implementation is usually harder
Usage
1. Create/locate a shared memory region M *shmid = shmget(IPC_PRIVATE, 40, IPC_CREAT | 0600);*
2. Attach M to process memory space *shm = (int*) shmat(shmid, NULL, 0)*
3. Read from/Write to M (Values written visible to all process that share M)
4. Detach from memory space after use *shmdt((char*) shm);*
5. Destroy *shmctl(shmid, IPC_RMID, 0);*
Only one process need to do this
Can only destroy if M is not attached to any process
Direct Communication
Send(p2, msg)
Receive(P1, msg)
Need to know pid to send (which is not fixed). Instead, send to MailBox, which can be shared among processes

Blocking Primitives (Synchronous):
- Send(): Sender is blocked until the message is received
- Receive(): Receiver is blocked until a message has arrived
Non-Blocking Primitives (async):
- Send(): Sender resume operation immediately
- Receiver(): Receiver either receive the message if available or some indication that message is not ready yet
Advantages:
- Portable: easily implemented on different processing environments
- Easier sync when synchronous primitive is used
Disadvantages:
- Inefficient: usu require OS
- Harder to use: limit msg size and format
Pipes
Pipe functions as circular bounded byte buffer with implicit synchronization:
- Writers wait when buffer is full
- Readers wait when buffer is empty
int pipe(int fd[])
fd[0]: reading end, fd[1]: writing end

Pipe Code
```
int pipeFd[2], pid, len;
char buf[100], *str = "Hello There!";
pipe( pipeFd );
if ((pid = fork()) > 0) { /* parent */
   close(pipeFd[READ_END]);
   write(pipeFd[WRITE_END], str,
strlen(str)+1);
   close(pipeFd[WRITE_END]);
} else {   /* child */
   close(pipeFd[WRITE_END]);
   len = read(pipeFd[READ_END], buf,
sizeof buf);
   printf("Proc %d read: %s\n", pid, buf);
   close(pipeFd[READ_END]);
}
```

Signals
A form of inter-process communication
- An asynchronous notification regarding an event
- Sent to a process/thread
- The recipient of the signal must handle the signal by:
- A default set of handlers OR
- User supplied handler (for some only)
Signal Code
```
void myOwnHandler( int signo ) {
if (signo == SIGSEGV){
   printf("Memory access blows up!\n");
   exit(1); }
}
int main(){
   int *ip = NULL;
   if (signal(SIGSEGV, myOwnHandler) ==
SIG_ERR)
   printf("Failed to register handler\n");
   *ip = 123;
   return 0;
}
```

Input Redirection
```
   int fp_in = open("./file.txt",
O_RDONLY);
   int fp_out = open("./talk.out",
O_CREAT | O_WRONLY);

  if(fork() == 0) {
     dup2(fp_in, STDIN_FILENO);
     dup2(fp_out, STDOUT_FILENO);
     execlp("./talk", "talk", (char *) 0);
     close(fp_in);
     close(fp_out);
  }
  else
     wait(NULL);
```

To simply redirect output from stdout:
```
close(STDOUT);
open(“result.txt”, O_WRONLY |
O_CREAT | O_TRUNC, 0644);
printf(“hello world”) // will be written
into result.txt file
```

PIPE
```
   int p[2];
   char str[] = "Hello this is the parent.";
   // This creates a pipe. p[0] is the
reading end,
   // p[1] is the writing end.
   if(pipe(p) < 0)
      perror("lab2p2e: ");
   // We will send a message from father
to child
   if(fork() != 0) {
      // Parent
      close(p[0]); // The the end we are not
using.
      write(p[1], str, strlen(str));
      close(p[1]);
      wait(NULL);
   }
   else
   {
      // Child
      char buffer[128];
      close(p[1]); // Close the writing end
      read(p[0], buffer, 127);
      printf("Child got the message
\"%s\"\n", buffer);
      close(p[0]);
   }
```