CS2106: Introduction to Operating System - AY2024/25, Semester 2 (Finals)

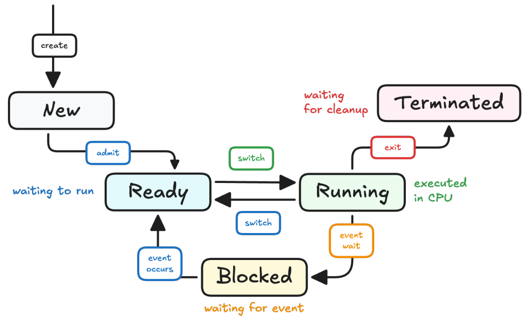
zaidan s. @zaidansani

Process Management

Stack frame structure:
Local variables, parameters, return PC, other info

Setup/Teardown stack frame:
1. Caller passes parameters with register and/or stack
2. Caller saves return PC on stack
3. Callee saves registers used, old FP, SP
4. Callee allocates space (callee local variables) stack.
5. Callee adjusts SP to point to new stack top.
6. Execution of call
7. Callee restores saved registers, FP, SP
8. Callee places return result on stack
9. Callee restores saved SP
10. Caller utilises return result
11. Caller continues execution

5-State Process Model



Syscall mechanism
1. User program: library call
2. Library call: syscall no. of designated loc.
3. Library call: TRAP instruction to kernel mode.
4. System call handler is determined using syscall.
5. Syscall handler is executed.
6. Control return to lib call, switch back to user mode.
7. Library call returns to user program.

Process Control Block: information about a process (registers, memory region information, PID, process state)

Exception vs Interrupt:
Synchronous vs asynchronous, executes exception handler vs executes interrupt handler

Abstractions in UNIX
pid_t fork(void): Returns: Parent: **child PID**, child: **0**.
int exec *(....) : Replace code with another
int wait(int *status): Waits for process to end

Process Scheduling

Non-preemptive: Process stays scheduled
Preemptive: Process is suspended at time quota

Batch processing
Turnaround time Total time from arrival to finish
Throughput Number of tasks/unit time

CPU utilisation % of time CPU is doing work

Scheduling algorithms
FCFS: FIFO queue, **No starvation**
SJF: Smallest CPU time, **Possible starvation**, predicts using exponential average of history.
SRT: Preemptive SJF

Convoy effect Fight for CPU and fight for I/O together

Interactive environment
Response time: Time between request & response

Scheduling algorithms
RR: FCFS, preemptive
Priority: Each task gets priority, highest first
preemptive: new process preempts, non-preemptive: wait for next round. can result in starvation
Priority inversion: higher priority task forced to block
MLFQ:
p(A) > p(B), A runs — p(A) == p(B), RR.
New job gets highest priority. If it uses time slice fully, priority **reduced**. Else, **retained**.
Lottery: Tickets assigned, randomly chosen winner

Threads

User thread: Implemented as user library
More flexible, kernel unaware

Kernel thread: Implemented in OS
Thread level scheduling possible, slower, less flexible

Hybrid thread: Both

```
int pthread_create(pthread_t *thread,
  const pthread_attr_t * attr,
  void *(*start_routine) (void(*)), void *arg);
```

```
int pthread_exit(void *retval);
int pthread_join(pthread_t thread, void **retval)
```

Inter-Process Communication

Shared memory
Efficient, easy to use, synchronisation, harder to implement

```
int shmget(key_t key, size_t size, int shmflg)
void *shmat(int shmid, const void *shmaddr,
  int shmflg, int shmdt)
int shmdt(const void *shmaddr)
int shmctl(int shmid, int cmd, struct shmid_ds *buf)
```

Message parsing
Portable, easier sync, inefficient, harder to use

Direct One buffer per pair of (sender, receiver)
Indirect Send/receive to port/mailbox

Blocking Until message received/arrives
Non-blocking (Async) Does not block

Unix Pipes Fixed size circular byte buffer
Writer wait when buffer full, reader wait when empty

```
int pipe(int pipefd[2])
pipefd[0] \\ read, pipefd[1] \\ write
```

Synchronisation

Properties
Mutual exclusion If process in CS, all other processes prevented from entering
Progress No process in CS, one waiting process should be granted access
Bounded wait Process requests to enter, there should be an upper bound on amount of processes that can enter before it
Independence Process not in CS should not block other process

Challenges
Deadlock All processes blocked
Livelock Processes keep changing state to avoid deadlock
Starvation Processes never get to make progress

Critical Section Implementations
Global Lock = 0
Busy waiting, Interrupt disabling, busy waiting
Turn-based Want = int[2]
No deadlocks if both Wants are not 1, deadlock otherwise
Peterson's Both global and turn-based locks
Assumes turn is atomic.
Busy waiting

Semaphore