SE350 Final Cheat Sheet

Pre-Midterm Material

OS Definition

OS Roles:

- Referee: resource allocation, isolation, and communication b/w Applications, Users.
- Illusionist: Applications appear to have entire machine; infinite cores, (near) infinite memory, reliable storage and network.
- Glue: System Libraries; Hardware Abstraction

OS Challenges: Reliability & Availability; Security & Privacy; Performance

I/O

Device I/O: Memory Mapped (shared space between DRAM and I/O); Port-Mapped.
Programmed I/O: CPU waits for I/O
Interrupt-Driven I/O: CPU pokes for request, device sends interrupt when done.
DMA: No buffer needed for I/O R/W.
Faster DMA: Buffer Descriptor, Queue of I/O Requests

Other Definitions

PCB: Stores where process is stored in memory, where executable image is, which user runs it, privileges
Hardware Support for Dual Mode: Privileged instr, timer interrupt, memory protection(base & bounds, virtual)
Base & bounds flaws: Fixed heap/stack, no memory sharing, fragmentation, no relative memory addresses
Switching safely: Limited entry (interrupt vector), atomic transfer of control, transparent restartable execution
Reasons to switch to kernel: exception, interrupt, system call, polling
Reasons to switch to User: new process, resume, switch process, user-level upcall
Interrupt Stack: Store registers, Frame pointer, locals, and return address. Kernel stack for each process
Switch Steps: Save SP, execution flags, and inst pointer; Switch onto kernel exception stack; Push those 3 values onto new stack; Optionally save error code; Invoke interrupt handler.

Thread States: init, ready, waiting, running, finished
Preemptive Thread: Can switch anytime.
Cooperative: run without interrupt, explicitly give up CPU; long-running threads can monopolize processor (starvation, non-responsiveness)
Data Stored in TCB: Stack info, saved registers, metadata.
Shared State: Heap, global vars, code **Thread Context Switch:** copy current thread registers from processor to TCB. Copy new thread registers from TCB to processor. Save old threads stack pointer.
Multithreaded Processes:

1. user = kernel thread, kernel does switching.
2. green threads, user level library that does switches. (bad: appears as one process to the kernel, not efficient).
3. scheduler activations, kernel gives processor to user lib, thread lib does switch and scheduling.

Safety: Never enter bad state. **Liveness:** Eventually enters good state.
Shared Objects: can be accessed safely by multiple threads. Has synchronization variables (locks)
Lock: synchronization var that provides mutual exclusion
Condition Variables (CV): a sync object that lets thread efficiently wait for a change in shared state that is protected by lock. (always use in a loop). Memoryless.
Spinlocks: for multiprocessor. Processor waits in loop for lock to become free. (low overhead if held briefly, less than context switch). Deadlock can happen unless all interrupts are disabled.
Semaphore: non negative int val. `P()`: wait for val ¿ 0, then val–. `V()`: val++, wakes up waiters. Can use like a lock. Better for async IO comm.
Structured Sync: add locks to shared objects. Wait in loop. Use signal/broadcast. Leave shared vars in consistent state.
Uniprocessor Locks: implement by temporarily disable/enable interrupts when acquiring/releasing lock. Move threads to WAITING queue if lock is busy.
Multiprocessor Locks: disable/enable interrupts is not enough. Need atomic read-modify-write instruction, will execute atomically to all other processors (`test_and_set instr`). Use this to implement spin locks.

Readers/Writers Lock: one writer if no readers. Many readers if no writer. kirito = waitpid(pid) or just wait(&kirito).
Process: instance of program that is running.
Thread: a *single execution sequence* that represents a *separately schedulable* task
Shell: job control system
Event driven: single thread with event queue.
Multithread: create new thread for each event

Implementations

Synchronization

Uniprocessor Lock

```
Lock::acquire() {
  disableInterrupts();
  if (value == BUSY) {
    waiting.add(myTCB);
    myTCB->state = WAITING;
    next = readyList.remove();
    thread_switch(myTCB, next);
    myTCB->state = RUNNING;
  } else {
    value = BUSY;
  }
  enableInterrupts();
}
```

```
Lock::release() {
  disableInterrupts();
  if (!waiting.Empty()) {
    next = waiting.remove();
    next->state = READY;
    readyList.add(next);
  } else {
    value = FREE;
  }
  enableInterrupts();
}
```

In a Uniprocessor machine, simply need to store TCB of current thread in a global variable

Multiprocessor Lock

```
Lock::acquire() {
  spinLock.acquire();
  if (value == BUSY) {
    waiting.add(myTCB);
    scheduler.suspend(&spinlock);
    // scheduler releases spinlock
  } else {
    value = BUSY;
    spinLock.release();
  }
}
```

```
Lock::release() {
  spinLock.acquire();
  if (!waiting.Empty()) {
    next = waiting.remove();
    scheduler.makeReady(next);
  } else {
    value = FREE;
  }
  spinLock.release();
}
```

```
Sched::suspend(SpinLock *lock) {
  TCB *next;
  disableInterrupts()
  schedSpinLock.acquire();
  spinLock->release();
  myTCB->state = WAITING;
  next = readyList.remove();
  thread_switch(myTCB, next);
  myTCB->state = RUNNING;
  schedSpinLock.release();
  enableInterrupts();
}
```

```
Sched::makeReady(TCB *thread) {
  disableInterrupts();
  schedSpinLock.acquire();
  readyList.add(thread);
  thread->state = READY;
  schedSpinLock.release();
  enableInterrupts();
}
```